COURSENOTES

CS2604:
Data Structures and File Processing

Java Edition

Clifford A. Shaffer
Department of Computer Science
Virginia Tech

The Need for Data Structures

Data structures organize data
$\Rightarrow$ **more efficient programs**.

- More powerful computers $\Rightarrow$ more complex applications.
- More complex applications demand more calculations.
- Complex computing tasks are unlike our everyday experience.

Any organization for a collection of records can be searched, processed in any order, or modified.

- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be **efficient** if it solves the problem within its **resource constraints**.

- space
- time

The **cost** of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some questions to ask:

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?

- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

Goals of this Course

1. Reinforce the concept that there are costs and benefits for every data structure.

2. Learn the commonly used data structures. These form a programmer's basic data structure "toolkit."

3. Understand how to measure the effectiveness of a data structure or program.
   - These techniques also allow you to judge the merits of new data structures that you or others might invent.

Definitions

A **type** is a set of values.

A **data type** is a type and a collection of operations that manipulate the type.

A **data item** or **element** is a piece of information or a record.

A data item is said to be a **member** of a data type.

A **simple data item** contains no subparts.

An **aggregate data item** may contain several pieces of information.

Abstract Data Types

Abstract Data Type (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

Each ADT operation is defined by its inputs and outputs.

Encapsulation: hide implementation details

A **data structure** is the physical implementation of an ADT.

- Each operation associated with the ADT is implemented by one or more subroutines in the implementation.

Data structure usually refers to an organization for data in main memory.

File structure: an organization for data on peripheral storage, such as a disk drive or tape.

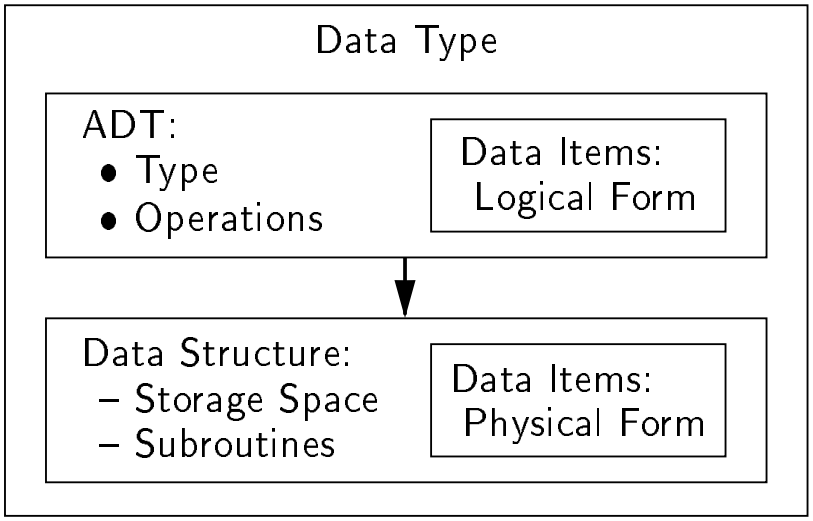
An ADT manages complexity through abstraction: **metaphor**.

Logical vs. Physical Form

Data items have both a **logical** and a **physical** form.

Logical form: definition of the data item within an ADT.

Physical form: implementation of the data item within a data structure.



Problems

Problem: a task to be performed.

- Best thought of as inputs and matching outputs.
- Problem definition should include constraints on the resources that may be consumed by any acceptable solution.

Problems $\Leftrightarrow$ mathematical functions

- A **function** is a matching between inputs (the **domain**) and outputs (the **range**).
- An **input** to a function may be single number, or a collection of information.
- The values making up an input are called the **parameters** of the function.
- A particular input must always result in the same output every time the function is computed.

Algorithms and Programs

Algorithm: a method or a process followed to solve a problem.

An algorithm takes the input to a problem (function) and transforms it to the output.

A problem can have many algorithms.

An algorithm possesses the following properties:

1. It must be **correct**.
2. It must be composed of a series of **concrete steps**.
3. There can be **no ambiguity** as to which step will be performed next.
4. It must be composed of a **finite** number of steps.
5. It must **terminate**.

A **computer program** is an instance, or concrete representation, for an algorithm in some programming language.

Mathematical Background

Set concepts and notation

Recursion

Induction proofs

Logarithms

Summations

Estimation Techniques

Known as "back of the envelope" or "back of the napkin" calculation.

1. Determine the major parameters that affect the problem.
2. Derive an equation that relates the parameters to the problem.
3. Select values for the parameters, and apply the equation to yield an estimated solution.

Example:

How many library bookcases does it take to store books totaling one million pages?

Estimate:

- pages/inch
- feet/shelf
- shelves/bookcase

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

1. Empirical comparison (run programs).

2. Asymptotic Algorithm Analysis.

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on "size" of the input.

Running time is expressed as $\mathbf{T}(n)$ for some function $\mathbf{T}$ on input size n.

Examples of Growth Rate

Example 1:

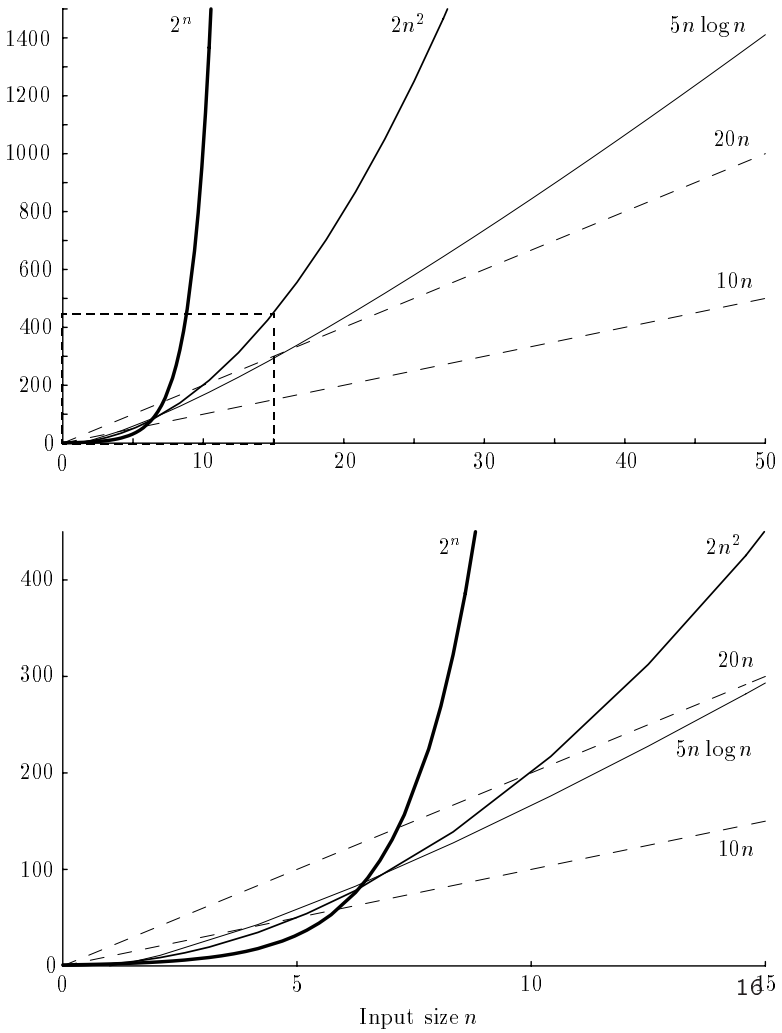
```
static int largest(int[] array) { // Find largest val
  int currLargest = 0;            // Store largest val
  for (int i=0; i<array.length; i++) // For each elem
   if (array[i] > currLargest)    //   if largest
      currLargest = array[i];     //     remember it
  return currLargest;             // Return largest val
}
```

Example 2: Assignment statement

Example 3:

```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum++;
```

Growth Rate Graph



Best, Worst and Average Cases

Not all inputs of a given size take the same time.

Sequential search for K in an array of n integers:

- Begin at first element in array and look at each element in turn until K is found.

Best Case:

Worst Case:

Average Case:

While average time seems to be the fairest measure, it may be difficult to determine.

When is worst case time important?

Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster?

$\mathbf{T}(n)$	n	n'	Change	n'/n
$10n$	1,000	10,000	$n' = 10n$	10
$20n$	500	5,000	$n' = 10n$	10
$5n \log n$	250	1,842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

n: Size of input that can be processed in one hour (10,000 steps).

n': Size of input that can be processed in one hour on the new machine (100,000 steps).

Asymptotic Analysis: Big-oh

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set O($f(n)$) if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

Usage: The algorithm is in O(n^2) in [best, average, worst] case.

Meaning: For *all* data sets big enough (i.e., $n > n_0$), the algorithm *always* executes in less than $cf(n)$ steps [in best, average or worst case].

Upper Bound.

Example: if $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in O(n^2).

Wish tightest upper bound:
While $\mathbf{T}(n) = 3n^2$ is in O(n^3), we prefer O(n^2).

Big-oh Example

Example 1. Finding value X in an array.

$\mathbf{T}(n) = c_s n/2$.
For all values of $n > 1$, $c_s n/2 \leq c_s n$.
Therefore, by the definition, $\mathbf{T}(n)$ is in $O(n)$ for $n_0 = 1$ and $c = c_s$.

Example 2. $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in average case

$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$.

$\mathbf{T}(n) \leq cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the definition.

Example 3: $\mathbf{T}(n) = c$. We say this is in $O(1)$.

Big-Omega

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.

Meaning: For *all* data sets big enough (i.e., $n > n_0$), the algorithm *always* executes in more than $cg(n)$ steps.

Lower Bound.

Example: $\mathbf{T}(n) = c_1 n^2 + c_2 n$.

$c_1 n^2 + c_2 n \geq c_1 n^2$ for all $n > 1$.
$\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

Want greatest lower bound.

Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

Simplifying Rules:

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Running Time of a Program

Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

Example 3:

```
sum = 0;
for (j=1; j<=n; j++)     // First for loop
   for (i=1; i<=j; i++)  //   is a double loop
      sum++;
for (k=0; k<n; k++)      // Second for loop
   A[k] = k;
```

More Examples

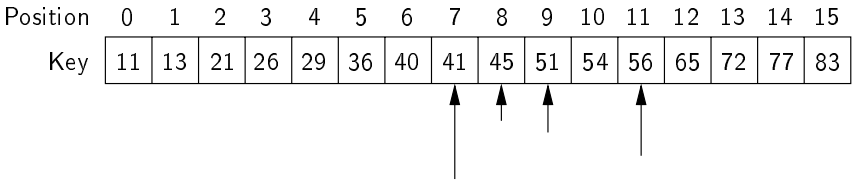
Example 4.

```
sum1 = 0;
for (i=1; i<=n; i++)      // First double loop
   for (j=1; j<=n; j++)   //   do n times
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)      // Second double loop
   for (j=1; j<=i; j++)   //   do i times
      sum2++;
```

Example 5.

```
sum1 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=k; j++)
      sum2++;
```


Binary Search




```
static int binary(int K, int[] array,
                          int left, int right) {
  // Return position in array (if any) with value K
  int l = left-1;
  int r = right+1;   // l and r are beyond array bounds
  while (l+1 != r) { // Stop when l and r meet
    int i = (l+r)/2; // Look at middle of subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```

Analysis: How many elements can be examined in the worst case?

Other Control Statements

`while` loop: analyze like a `for` loop.

`if` statement: Take greater complexity of then/else clauses.

`switch` statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.

Analyzing Problems

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for *every possible algorithm*.

Example: Sorting

1. Cost of I/O: $\Omega(n)$
2. Bubble or insertion sort: $O(n^2)$
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$
4. We prove later that sorting is $\Omega(n \log n)$

Multiple Parameters

Compute the rank ordering for all C pixel values in a picture of P pixels.

```
for (i=0; i<C; i++)   // Initialize count
   count[i] = 0;
for (i=0; i<P; i++)   // Look at all of the pixels
   count[value(i)]++; // Increment proper value count
sort(count);          // Sort pixel value counts
```

If we use P as the measure, then time is $\Theta(P \log P)$.

More accurate is $\Theta(P + C \log C)$.

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm

Space: Data Structure

Space/Time Tradeoff Principle:

One can often achieve a reduction in time is one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  Boolean flags
- Table lookup
  Factorials

Disk Based Space/Time Tradeoff Principle:

The smaller you can make your disk storage requirements, the faster your program will run.

Lists

A **list** is a finite, ordered sequence of data items called **elements**.

Each list element has a data type.

The **empty** list contains no elements.

The **length** of the list is the number of elements currently stored.

The beginning of the list is called the **head**, the end of the list is called the **tail**.

Sorted lists have their elements positioned in ascending order of value, while **unsorted lists** have no necessary relationship between element values and positions.

Notation: (a_0, a_1, ..., a_{n-1})

What operations should we implement?

List ADT

```
interface List {                    // List ADT
public void clear();                // Remove all Objects
public void insert(Object item);    // Insert at curr pos
public void append(Object item);    // Insert at tail
public Object remove();             // Remove/return curr
public void setFirst();             // Set to first pos
public void next();                 // Move to next pos
public void prev();                 // Move to prev pos
public int length();                // Return curr length
public void setPos(int pos);        // Set curr position
public void setValue(Object val); // Set current value
public Object currValue();          // Return curr value
public boolean isEmpty();           // True if empty list
public boolean isInList();          // True if in list
public void print();                // Print all elements
} // interface List
```

List ADT Examples

List: (12, 32, 15)

```
MyLst.insert(element);
```

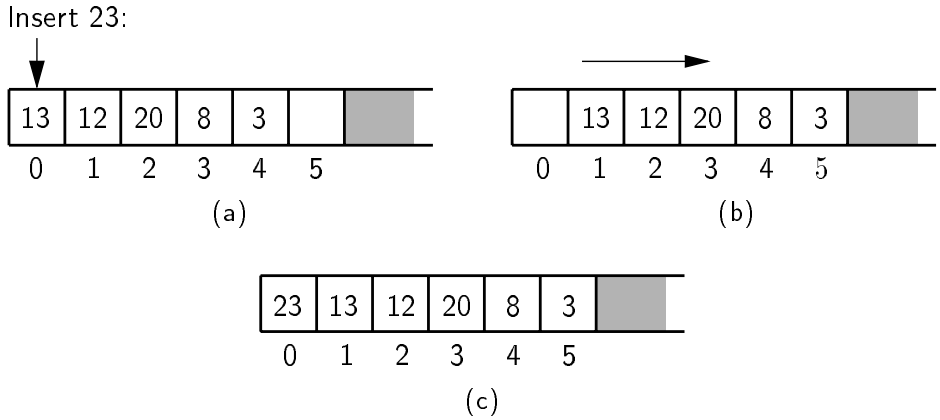
Assume `MyPos` has 32 as current element:

```
MyLst.insert(99);
```

Process an entire list:

```
for (MyLst.setFirst(); MyLst.isInList(); MyLst.next())
  DoSomething(MyLst.currValue());
```


Array-Based List Insert



Array-Based List Class

```
class AList implements List {   // Array-based list

private static final int defaultSize = 10;

private int msize;               // Maximum size of list
private int numInList;           // Actual list size
private int curr;                // Position of curr
private Object[] listArray;      // Array holding list

AList() { setup(defaultSize); } // Constructor
AList(int sz) { setup(sz); }    // Constructor

private void setup(int sz) {    // Do initializations
  msize = sz;
  numInList = curr = 0;
  listArray = new Object[sz];   // Create listArray
}

public void clear()     // Remove all Objects from list
{ numInList = curr = 0; // Simply reinitialize values

public void insert(Object it) { // Insert at curr pos
  Assert.notFalse(numInList < msize, "List is full");
  Assert.notFalse((curr >=0) && (curr <= numInList),
                  "Bad value for curr");
  for (int i=numInList; i>curr; i--) // Shift up
    listArray[i] = listArray[i-1];
  listArray[curr] = it;
  numInList++;                  // Increment list size
}
```


Array-Based List Class (cont)

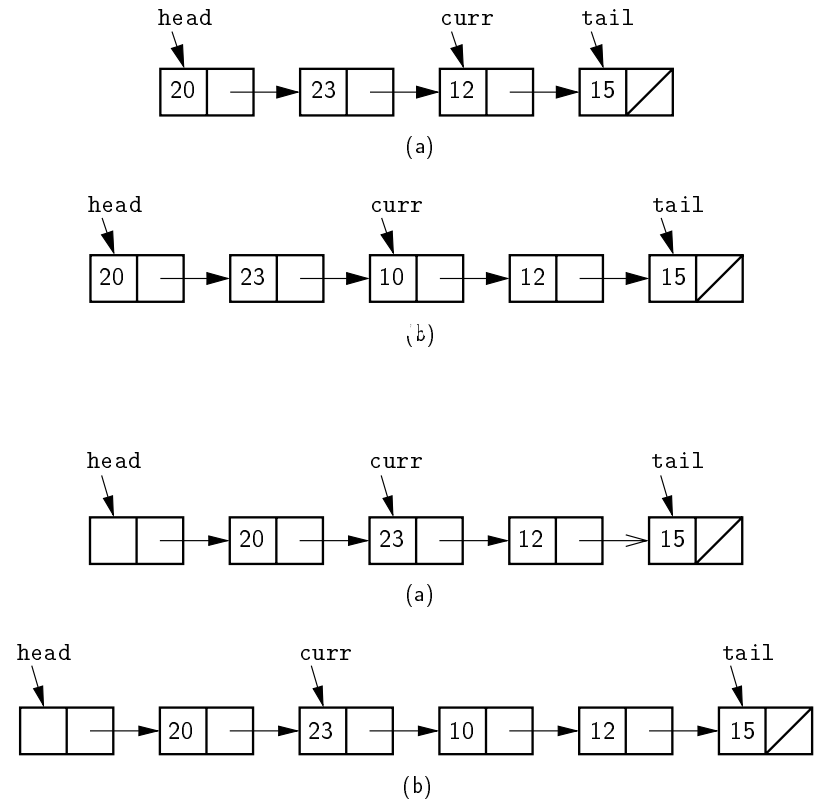
```
public void append(Object it) { // Insert at tail
  Assert.notFalse(numInList < msize, "List is full");
  listArray[numInList++] = it;  // Increment list size
}

public Object remove() { // Remove and return Object
  Assert.notFalse(!isEmpty(), "No delete: list empty");
  Assert.notFalse(isInList(), "No current element");
  Object it = listArray[curr];  // Hold removed Object
  for(int i=curr; i<numInList-1; i++) // Shift down
    listArray[i] = listArray[i+1];
  numInList--;                  // Decrement list size
  return it;
}

public void setFirst() { curr = 0; } // Set to first
public void prev()  { curr--; }  // Move curr to prev
public void next()  { curr++; }  // Move curr to next
public int length() { return numInList; }
public void setPos(int pos) { curr = pos; }
public boolean isEmpty() { return numInList == 0; }

public void setValue(Object it) { // Set current value
  Assert.notFalse(isInList(), "No current element");
  listArray[curr] = it;
}

public boolean isInList() // True if curr within list
  { return (curr >= 0) && (curr < numInList); }
} // Array-based list implementation
```

Link Class

Dynamic allocation of new list elements.

```
class Link {              // A singly linked list node
  private Object element; // Object for this node
  private Link next;      // Pointer to next node
  Link(Object it, Link nextval)          // Constructor
    { element = it;  next = nextval; }
  Link(Link nextval) { next = nextval; } // Constructor
  Link next() { return next; }
  Link setNext(Link nextval) { return next = nextval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }
}
```


Linked List Position

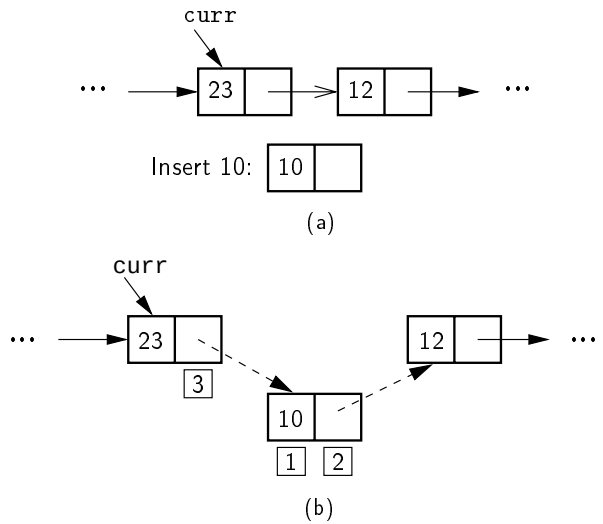


Linked List Implementation

```
public class LList implements List { // Linked list
private Link head;    // Pointer to list header
private Link tail;    // Pointer to last Object in list
protected Link curr;  // Position of current Object

LList(int sz) { setup(); }     // Constructor
LList() { setup(); }           // Constructor
private void setup()
  { tail = head = curr = new Link(null); }

public void setFirst() { curr = head; }
public void next()
  { if (curr != null) curr = curr.next(); }

public void prev() {       // Move to previous position
  Link temp = head;
  if ((curr == null) || (curr == head)) // No prev
    { curr = null;  return; }           //  so return
  while ((temp != null) && (temp.next() != curr))
        temp = temp.next();
  curr = temp;
}

public Object currValue() { // Return current Object
  if (!isInList()) return null;
  return curr.next().element();
}

public boolean isEmpty()    // True if list is empty
  { return head.next() == null; }
} // Linked list class
```

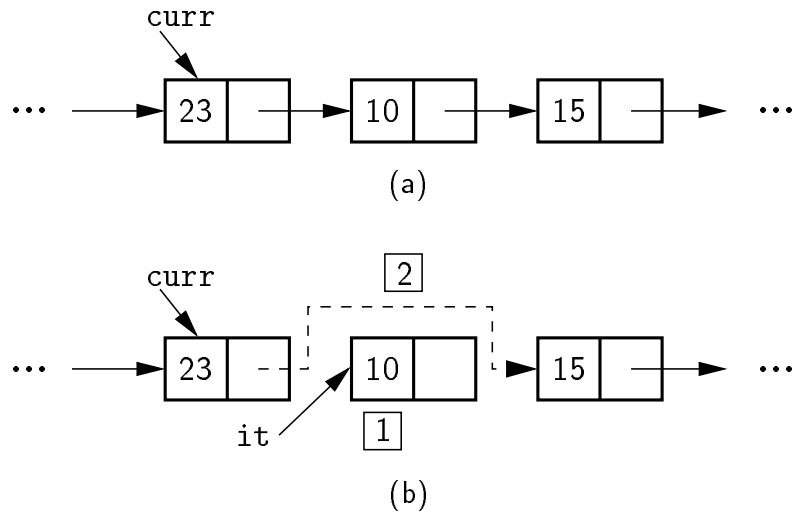

Linked List Insertion

```
// Insert Object at current position
public void insert(Object it) {
  Assert.notNull(curr, "No current element");
  curr.setNext(new Link(it, curr.next()));
  if (tail == curr)           // Appended new Object
    tail = curr.next();
}
```



Linked List Remove

```
public Object remove() { // Remove/return curr Object
  if (!isInList()) return null;
  Object it = curr.next().element(); // Remember value
  if (tail == curr.next()) tail = curr; // Set tail
  curr.setNext(curr.next().next());  // Cut from list
  return it;                         // Return value
}
```



Freelists

System **new** and garbage collection are slow.

```
class Link {  // Singly linked list node with freelist
  private Object element; // Object for this Link
  private Link next;      // Pointer to next Link
  Link(Object it, Link nextval)
  { element = it;  next = nextval; }
  Link(Link nextval) { next = nextval; }
  Link next() { return next; }
  Link setNext(Link nextval) { return next = nextval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }

  // Extensions to support freelists
  static Link freelist = null; // Freelist for class

  static Link get(Object it, Link nextval) {
    if (freelist == null)
      return new Link(it, nextval);
    Link temp = freelist;
    freelist = freelist.next();
    temp.setElement(it);
    temp.setNext(nextval);
    return temp;
  }

  void release() {
    element = null;  next = freelist;  freelist = this;
  }
}
```

Comparison of List Implementations

Array-Based Lists:

- Insertion and deletion are $\Theta(n)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:

- Insertion and deletion $\Theta(1)$;
  prev and direct access are $\Theta(n)$.
- Space grows with number of elements.
- Every element requires overhead.

Space "break-even" point:

$$DE = n(P + E); \quad n = \frac{DE}{P + E}$$

E: Space for data value

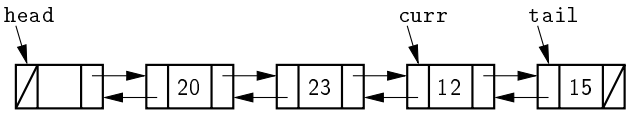
P: Space for pointer

D: Number of elements in array

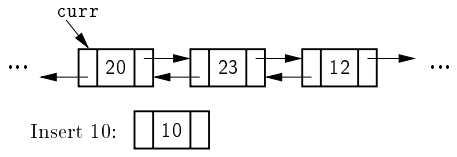
Doubly Linked Lists

Simplify insertion and deletion: Add a **prev** pointer.

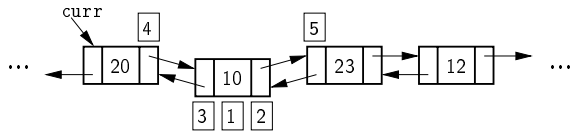
```
class DLink {              // A doubly-linked list node
  private Object element; // Object for this node
  private DLink next;     // Pointer to next node
  private DLink prev;     // Pointer to previous node
  DLink(Object it, DLink n, DLink p)
  { element = it;  next = n; prev = p; }
  DLink(DLink n, DLink p) { next = n;  prev = p; }
  DLink next() { return next; }
  DLink setNext(DLink nextval) { return next=nextval; }
  DLink prev() { return prev; }
  DLink setPrev(DLink prevval) { return prev=prevval; }
  Object element() { return element; }
  Object setElement(Object it) { return element = it; }
}
```



Doubly Linked List Operations



(a)



(b)

```
// Insert Object at current position
public void insert(Object it) {
  Assert.notNull(curr, "No current element");
  curr.setNext(new DLink(it, curr.next(), curr));
  if (curr.next().next() != null)
    curr.next().next().setPrev(curr.next());
  if (tail == curr)             // Appended new Object
    tail = curr.next();
}

public Object remove() { // Remove/return curr Object
  Assert.notFalse(isInList(), "No current element");
  Object it = curr.next().element(); // Remember Object
  if (curr.next().next() != null)
    curr.next().next().setPrev(curr);
  else tail = curr;  // Removed last Object: set tail
  curr.setNext(curr.next().next()); // Remove from list
  return it;                    // Return value removed
}
```


Stacks

LIFO: Last In, First Out

Restricted form of list: Insert and remove only at front of list.

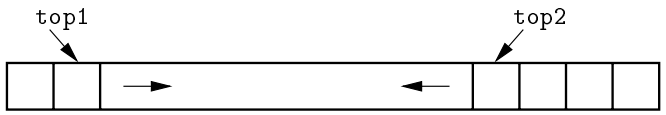
Notation:

- Insert: PUSH
- Remove: POP
- The accessible element is called TOP.

Array-Based Stack

Define `top` as first free position.

```
class Stack class AStack { // Array based stack class
  private static final int defaultSize = 10;
  private int size;            // Maximum size of stack
  private int top;             // Index for top Object
  private Object [] listarray; // Array holding stack
  AStack() { setup(defaultSize); }
  AStack(int sz) { setup(sz); }

  public void setup(int sz)
  { size = sz;  top = 0; listarray = new Object[sz]; }

  public void clear() { top = 0; } // Clear all Objects

  public void push(Object it) // Push onto stack
  { Assert.notFalse(top < size, "Stack overflow");
    listarray[top++] = it; }

  public Object pop()         // Pop Object from top
  { Assert.notFalse(!isEmpty(), "Empty stack");
    return listarray[--top]; }

  public Object topValue()    // Return top Object
  { Assert.notFalse(!isEmpty(), "Empty stack");
    return listarray[top-1]; }

  public boolean isEmpty() { return top == 0; }
};
```



Linked Stack

```
public class LStack {   // Linked stack class

private Link top;       // Pointer to list header

public LStack() { setup(); }        // Constructor
public LStack(int sz) { setup(); }  // Constructor

private void setup()     // Initialize stack
{ top = null; }          // Create header node

public void clear() { top = null; } // Clear stack

public void push(Object it) // Push Object onto stack
{ top = new Link(it, top); }

public Object pop() {    // Pop Object from top
  Assert.notFalse(!isEmpty(), "Empty stack");
  Object it = top.element();
  top = top.next();
  return it;
}

public Object topValue() // Get value of top Object
{ Assert.notFalse(!isEmpty(), "No top value");
  return top.element(); }

public boolean isEmpty() // True if stack is empty
{ return top == null; }
} // Linked stack class
```

Queues

FIFO: First In, First Out

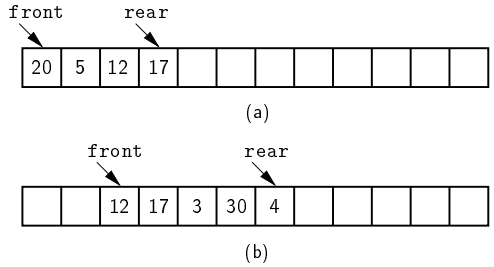
Restricted form of list:
Insert at one end, remove from other.

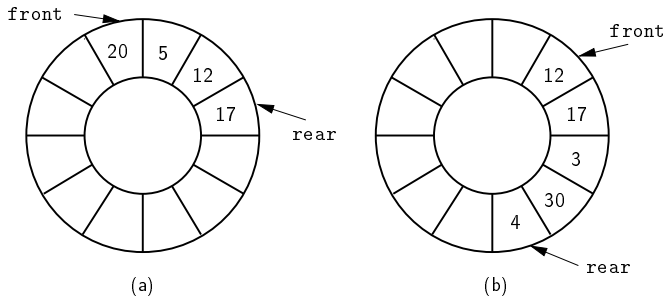
Notation:

- Insert: Enqueue
- Delete: Dequeue
- First element: FRONT
- Last element: REAR

Queue Implementations

Array-Based Queue



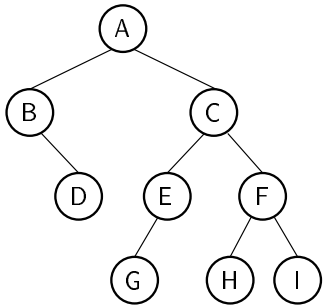


Linked Queue: modified linked list.

Binary Trees

A **binary tree** is made up of a finite set of nodes that is either **empty** or consists of a node called the **root** together with two binary trees, called the left and right **subtrees**, which are disjoint from each other and from the root.

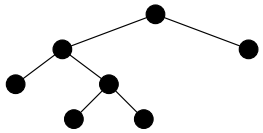
Notation: **Node**, **Children**, **Edge**, **Parent** **Ancestor**, **Descendant**, **Path**, **Depth**, **Height**, **Level**, **Leaf Node**, **Internal Node**, **Subtree**.



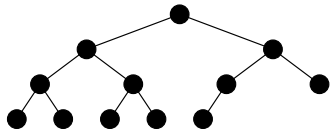
Full and Complete Binary Trees

Full binary tree: each node either is a leaf or is an internal node with exactly two non-empty children.

Complete binary tree: If the height of the tree is d, then all levels except possibly level d are completely full. The bottom level has all nodes to the left side.



(a)



(b)

Full Binary Tree Theorem

Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

Proof (by Mathematical Induction):

- **Base Case**: A full binary tree with 1 internal node must have two leaf nodes.
- **Induction Hypothesis**: Assume any full binary tree **T** containing $n-1$ internal nodes has n leaves.
- **Induction Step**: Given tree **T** with n internal nodes, pick internal node I with two leaf children. Remove I's children, call resulting tree **T'**. By induction hypothesis, **T'** is a full binary tree with n leaves. Restore i's two children. The number of internal nodes has now gone up by 1 to reach n. The number of leaves has also gone up by 1.

Full Binary Tree Theorem Corollary

Theorem: The number of `NULL` pointers in a non-empty binary tree is one more than the number of nodes in the tree.

Proof: Replace all null pointers with a pointer to an empty leaf node. This is a full binary tree.

Binary Tree Node ADT

```
interface BinNode { // ADT for binary tree nodes
  // Return and set the element value
  public Object element();
  public Object setElement(Object v);

  // Return and set the left child
  public BinNode left();
  public BinNode setLeft(BinNode p);

  // Return and set the right child
  public BinNode right();
  public BinNode setRight(BinNode p);

  // Return true if this is a leaf node
  public boolean isLeaf();
} // interface BinNode
```


Traversals

Any process for visiting the nodes in some order is called a **traversal**.

Any traversal that lists every node in the tree exactly once is called an **enumeration** of the tree's nodes.

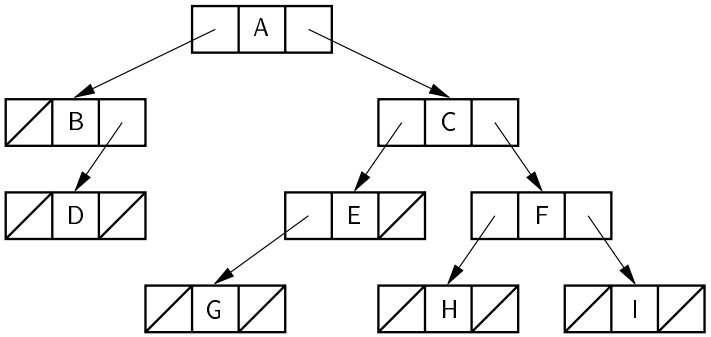
Preorder traversal: Visit each node *before* visiting its children.

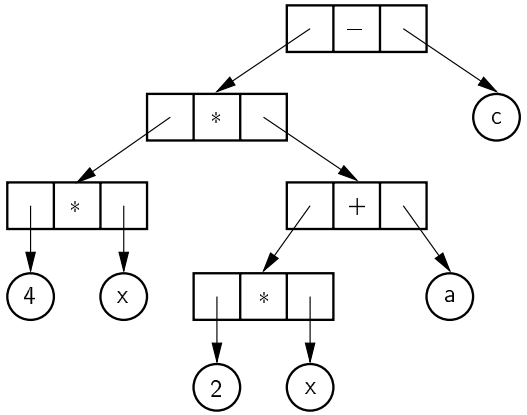
Postorder traversal: Visit each node *after* visiting its children.

Inorder traversal: Visit the left subtree, then the node, then the right subtree.

```
void preorder(BinNode rt) // rt is root of subtree
{
  if (rt == null) return; // Empty subtree
  visit(rt);
  preorder(rt.left());
  preorder(rt.right());
}
```

Binary Tree Implementation





Inheritance

```
class LeafNode implements BinNode { // Leaf node
  private String var;               // Operand value

  public LeafNode(String val) { var = val; }
  public Object element() { return var; }
  public Object setElement(Object v)
    { return var = (String)v; }
  public BinNode left() { return null; }
  public BinNode setLeft(BinNode p) { return null; }
  public BinNode right() { return null; }
  public BinNode setRight(BinNode p) { return null; }
  public boolean isLeaf() { return true; }
} // class LeafNode

class IntlNode implements BinNode { // Internal node
  private BinNode left;             // Left child
  private BinNode right;            // Right child
  private Character opx;            // Operator value

  public IntlNode(Character op, BinNode l, BinNode r)
    { opx = op; left = l; right = r; } // Constructor
  public Object element() { return opx; }
  public Object setElement(Object v)
    { return opx = (Character)v; }
  public BinNode left() { return left; }
  public BinNode setLeft(BinNode p) {return left = p;}
  public BinNode right() { return right; }
  public BinNode setRight(BinNode p)
    { return right = p; }
  public boolean isLeaf() { return false; }
} // class IntlNode
```

Inheritance (cont)

```
static void traverse(BinNode rt) { // Preorder
  if (rt == null) return;          // Nothing to visit
  if (rt.isLeaf())                 // Do leaf node
    System.out.println("Leaf: " + rt.element());
  else {                           // Do internal node
    System.out.println("Internal: " + rt.element());
    traverse(rt.left());
    traverse(rt.right());
  }
}
```


Space Overhead

From Full Binary Tree Theorem:

Half of pointers are `NULL`.

If leaves only store information, then overhead depends on whether tree is full.

All nodes the same, with two pointers to children:

Total space required is $(2p + d)n$.

Overhead: $2pn$.

If $p = d$, this means $2p/(2p + d) = 2/3$ overhead.

Eliminate pointers from leaf nodes:

$$\frac{\frac{n}{2}(2p)}{\frac{n}{2}(2p) + dn} = \frac{p}{p + d}$$

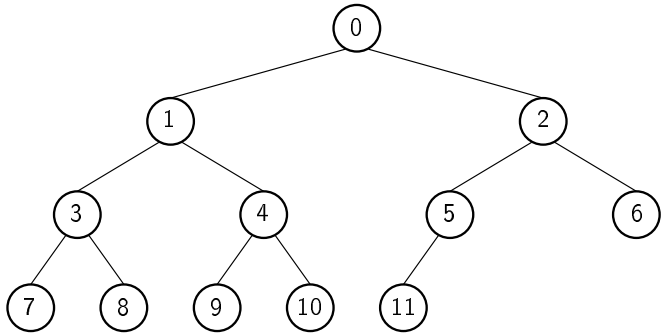
This is 1/2 if $p = d$.

$2p/(2p + d)$ if data only at leaves $\Rightarrow$ 2/3 overhead.

Some method is needed to distinguish leaves from internal nodes.

Array Implementation

For complete binary trees.



(a)

Node	0	1	2	3	4	5	6	7	8	9	10	11

- Parent(r) =
- Leftchild(r) =
- Rightchild(r) =
- Leftsibling(r) =
- Rightsibling(r) =

Huffman Coding Trees

ASCII Codes: 8 bits per character.

Fixed length coding.

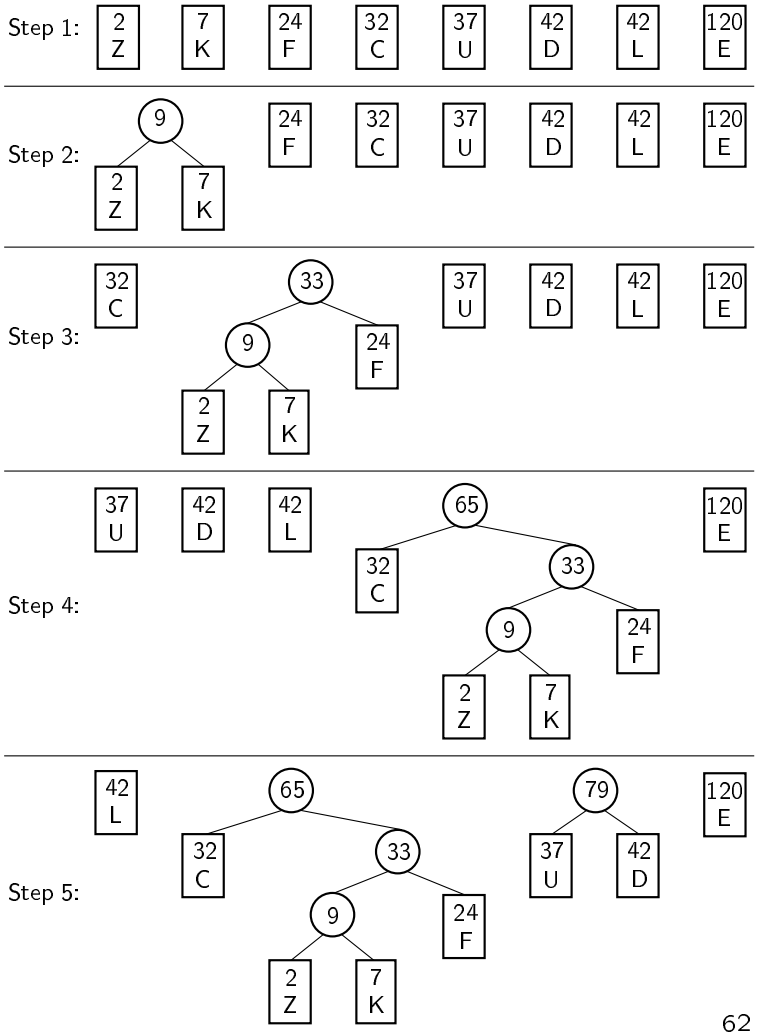
Can take advantage of relative frequency of letters to save space.

Variable length coding.

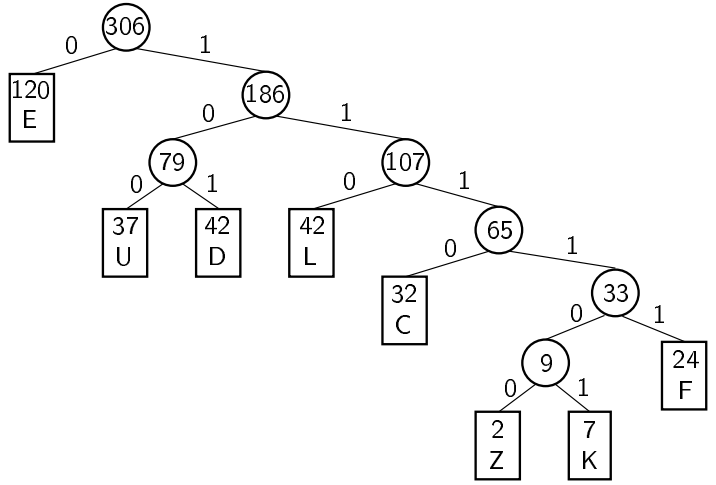
Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

Build the tree with **minimal external path weight**.

Huffman Tree Construction



Assigning Codes



Letter	Freq	Code	Bits
C	32		
D	42		
E	120		
F	24		
K	7		
L	42		
U	37		
Z	2		

Coding and Decoding

A set of codes are said to meet the **prefix property** if no code in the set is the prefix of another.

Code for DEED:

Decode 1011001110111101:

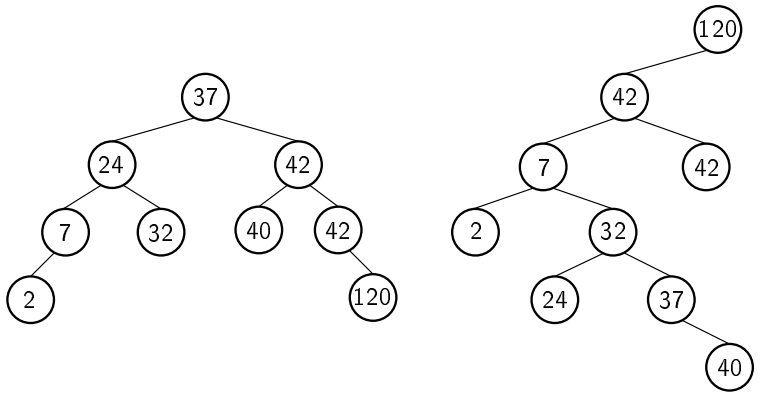
Expected cost per letter:

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
F	24	11111	5
K	7	111101	6
L	42	110	3
U	37	100	3
Z	2	111100	6

Binary Search Trees

Binary Search Tree (BST) Property

All elements stored in the left subtree of a node whose value is K have values less than K. All elements stored in the right subtree of a node whose value is K have values greater than or equal to K.



(a) (b)

BinNode Class

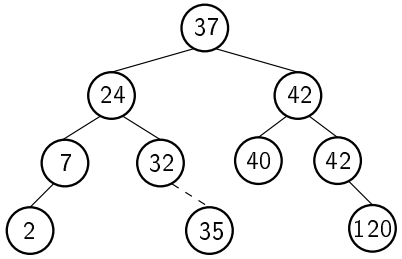
```
interface BinNode { // ADT for binary tree nodes
  // Return and set the element value
  public Object element();
  public Object setElement(Object v);

  // Return and set the left child
  public BinNode left();
  public BinNode setLeft(BinNode p);

  // Return and set the right child
  public BinNode right();
  public BinNode setRight(BinNode p);

  // Return true if this is a leaf node
  public boolean isLeaf();
} // interface BinNode
```


BST Search

```
public class BST { // Binary Search Tree implementation
  private BinNode root; // The root of the tree

  public BST() { root = null; } // Initialize root
  public void clear() { root = null; }
  public void insert(Elem val)
    { root = inserthelp(root, val); }
  public void remove(int key)
    { root = removehelp(root, key); }
  public Elem find(int key)
    { return findhelp(root, key); }
  public boolean isEmpty() { return root == null; }

  public void print() {
    if (root == null)
      System.out.println("The BST is empty.");
    else {
      printhelp(root, 0);
      System.out.println();
    }
  }

private Elem findhelp(BinNode rt, int key) {
  if (rt == null) return null;
  Elem it = (Elem)rt.element();
  if (it.key() > key)  return findhelp(rt.left(), key);
  else if (it.key() == key)  return it;
  else  return findhelp(rt.right(), key);
}
```

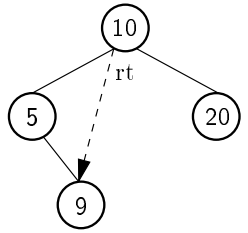
BST Insert

```
private BinNode inserthelp(BinNode rt, Elem val) {
  if (rt == null) return new BinNode(val);
  Elem it = (Elem) rt.element();
  if (it.key() > val.key())
    rt.setLeft(inserthelp(rt.left(), val));
  else
    rt.setRight(inserthelp(rt.right(), val));
  return rt;
}
```



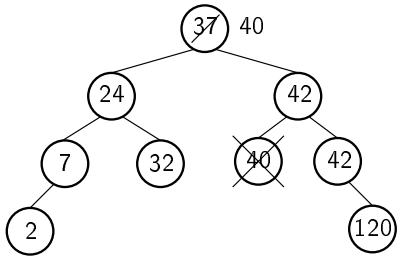
Remove Minimum Value

```
private BinNode deletemin(BinNode rt) {
  if (rt.left() == null)
    return rt.right();
  else {
    rt.setLeft(deletemin(rt.left()));
    return rt;
  }
}

private Elem getmin(BinNode rt) {
  if (rt.left() == null)
    return (Elem)rt.element();
  else return getmin(rt.left());
}
```



BST Remove

```
private BinNode removehelp(BinNode rt, int key) {
  if (rt == null) return null;
  Elem it = (Elem) rt.element();
  if (key < it.key())
    rt.setLeft(removehelp(rt.left(), key));
  else if (key > it.key())
    rt.setRight(removehelp(rt.right(), key));
  else {
    if (rt.left() == null)
      rt = rt.right();
    else if (rt.right() == null)
      rt = rt.left();
    else {
      Elem temp = getmin(rt.right());
      rt.setElement(temp);
      rt.setRight(deletemin(rt.right()));
    }
  }
  return rt;
}
```



Cost of BST Operations

Find:

Insert:

Remove:

Heaps

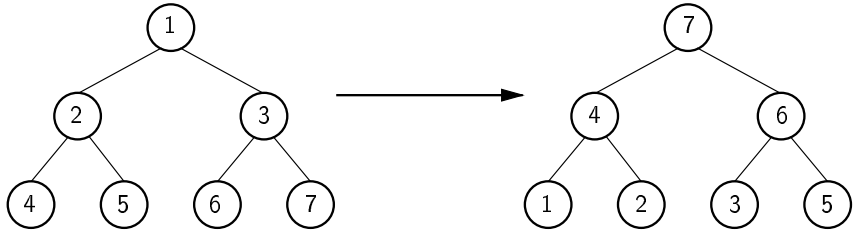
Heap: Complete binary tree with the **Heap Property**:

- Min-heap: all values less than child values.
- Max-heap: all values greater than child values.

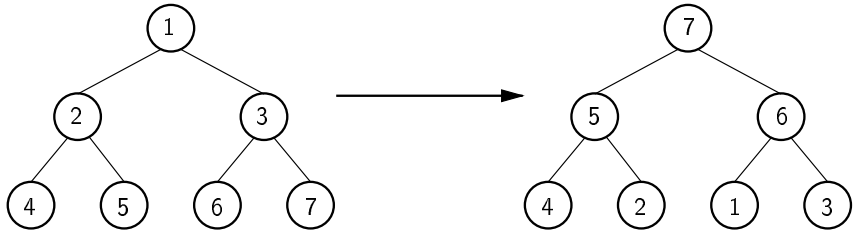
The values in a heap are **partially ordered**.

Heap representation: normally the array based complete binary tree representation.

Building the Heap



(a)



(b)

(a) requires exchanges (4-2), (4-1), (2-1), (5-2), (5-4), (6-3), (6-5), (7-5), (7-6).

(b) requires exchanges (5-2), (7-3), (7-1), (6-1).

Max Heap Implementation

```
public class MaxHeap {
private Elem[] Heap;  // Pointer to the heap array
private int size;     // Maximum size of the heap
private int n;        // Number of elements now in heap

public MaxHeap(Elem[] h, int num, int max)
{ Heap = h;  n = num;  size = max;  buildheap(); }

public int heapsize() // Return current size of heap
{ return n; }

public boolean isLeaf(int pos)  // TRUE if pos is leaf
{ return (pos >= n/2) && (pos < n); }

// Return position for left child of pos
public int leftchild(int pos) {
  Assert.notFalse(pos < n/2, "No left child");
  return 2*pos + 1;
}

// Return position for right child of pos
public int rightchild(int pos) {
  Assert.notFalse(pos < (n-1)/2, "No right child");
  return 2*pos + 2;
}

public int parent(int pos) { // Return pos for parent
  Assert.notFalse(pos > 0, "Position has no parent");
  return (pos-1)/2;
}
```


Siftdown

For fast heap construction:

- Work from high end of array to low end.
- Call **siftdown** for each item.
- Don't need to call **siftdown** on leaf nodes.

```
public void buildheap() // Heapify contents of Heap
  { for (int i=n/2-1; i>=0; i--) siftdown(i); }

private void siftdown(int pos) { // Put in place
  Assert.notFalse((pos >= 0) && (pos < n),
                  "Illegal heap position");
  while (!isLeaf(pos)) {
    int j = leftchild(pos);
    if ((j<(n-1)) && (Heap[j].key() < Heap[j+1].key()))
      j++; // j now index of child with greater value
    if (Heap[pos].key() >= Heap[j].key()) return;
    DSutil.swap(Heap, pos, j);
    pos = j;  // Move down
  }
}
```

Cost for heap construction:

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} \approx n.$$

Priority Queues

A priority queue stores objects, and on request releases the object with greatest value.

Example: Scheduling jobs in a multi-tasking operating system.

The **priority** of a job may change, requiring some reordering of the jobs.

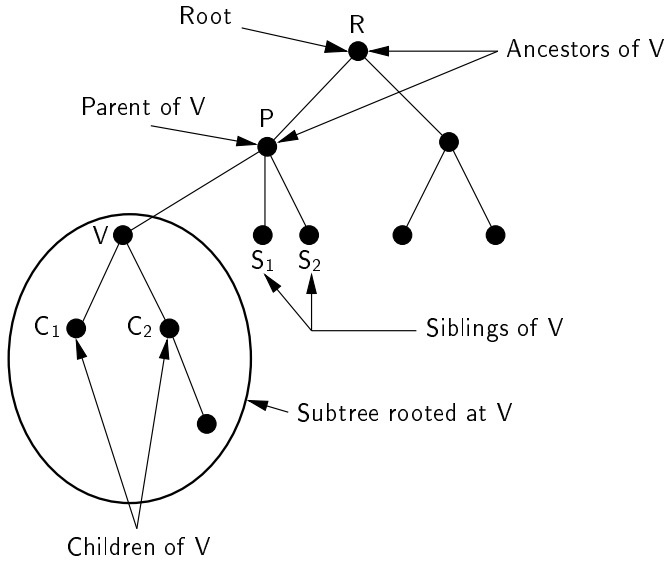
Implementation: use a heap to store the priority queue.

To support priority reordering, delete and re-insert. Need to know index for the object.

```
// Remove value at specified position
public Elem remove(int pos) {
  Assert.notFalse((pos >= 0) && (pos < n),
                  "Illegal heap position");
  DSutil.swap(Heap, pos, --n); // Swap with last value
  while (Heap[pos].key() > Heap[parent(pos)].key())
    DSutil.swap(Heap, pos, parent(pos)); // push up
  if (n != 0) siftdown(pos);             // push down
  return Heap[n];
}
```


General Trees

A **tree** T is a finite set of one or more nodes such that there is one designated node r called the root of T, and the remaining nodes in $(T - \{r\})$ are partitioned into $n \geq 0$ disjoint subsets T_1, T_2, ..., T_k, each of which is a tree, and whose roots r_1, r_2, ..., r_k, respectively, are children of r.

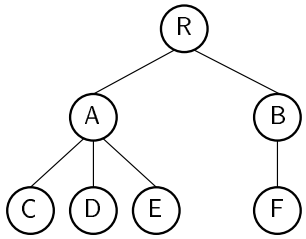


General Tree ADT

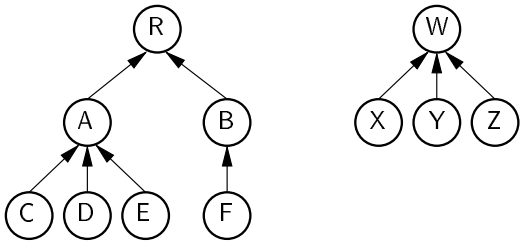
```
public interface GTNode {
  public Object value();
  public boolean isLeaf();
  public GTNode parent();
  public GTNode leftmost_child();
  public GTNode right_sibling();
  public void setValue(Object value);
  public void setParent(GTNode par);
  public void insert_first(GTNode n);
  public void insert_next(GTNode n);
  public void remove_first();
  public void remove_next();
}

public interface GenTree {
  public void clear();
  public GTNode root();
  public void newroot(Object value, GTNode first,
                                    GTNode sib);
}
```


General Tree Traversal

```
static void print(GTNode rt) { // Preorder traversal
  if (rt.isLeaf()) System.out.print("Leaf: ");
  else System.out.print("Internal: ");
  System.out.println(rt.value());
  GTNode temp = rt.leftmost_child();
  while (temp != null) {
    print(temp);
    temp = temp.right_sibling();
  }
}
```



Parent Pointer Implementation



Parent's Index		0	0	1	1	1	2		7	7	7
Label	R	A	B	C	D	E	F	W	X	Y	Z
Node Index	0	1	2	3	4	5	6	7	8	9	10

Parent Pointer representation is good for answering:

Are two elements in the same tree?

```
public boolean differ(int a, int b) {
  GTNode root1 = FIND(array[a]); // Find root of a
  GTNode root2 = FIND(array[b]); // Find root of b
  return root1 != root2;         // Compare roots
}
```


Equivalence Classes

When joining equivalence classes, want to keep depth small.

Weighted Union Rule: join the tree with fewer nodes to the tree with more nodes.

Limits depth to $\log n$ for n nodes.

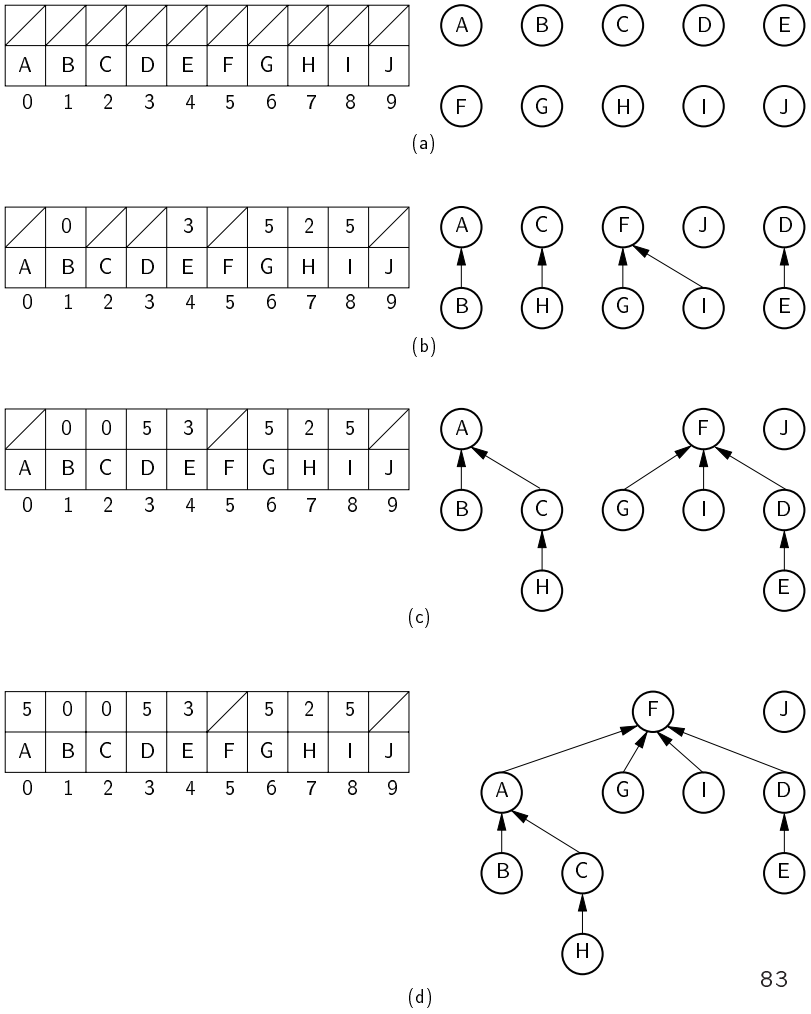
Path Compression: Make all nodes visited point to root.

```
class GTNode {   // General tree node for UNION/FIND
  private GTNode par;             // Parent pointer
  public GTNode() { par = null; } // Constuctor
  public GTNode parent() { return par; }
  public GTNode setParent(GTNode newpar)
  { return par = newpar; }
} // class GTNode
```

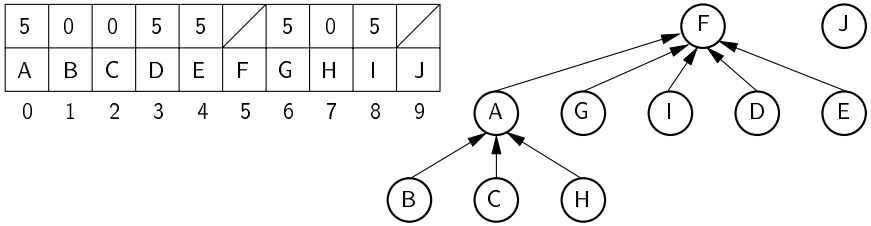
UNION-FIND

```
class GenTree {   // General Tree class for UNION/FIND
  private GTNode[] array;     // Node array

  public GenTree(int size) {  // Constructor
    array = new GTNode[size]; // Create node array
    for (int i=0; i<size; i++)
      array[i] = new GTNode();
  }

  public boolean differ(int a, int b) {
    GTNode root1 = FIND(array[a]);    // Find root of a
    GTNode root2 = FIND(array[b]);    // Find root of b
    return root1 != root2;            // Compare roots
  }

  public void UNION(int a, int b) {   // Merge subtrees
    GTNode root1 = FIND(array[a]);    // Find root of a
    GTNode root2 = FIND(array[b]);    // Find root of b
    if (root1 != root2) root2.setParent(root1);
  }

  private GTNode FIND(GTNode curr) {    // Find root
    while (curr.parent() != null) curr = curr.parent();
    return curr;                        // At root
  }
}
```


Equivalence Processing Example

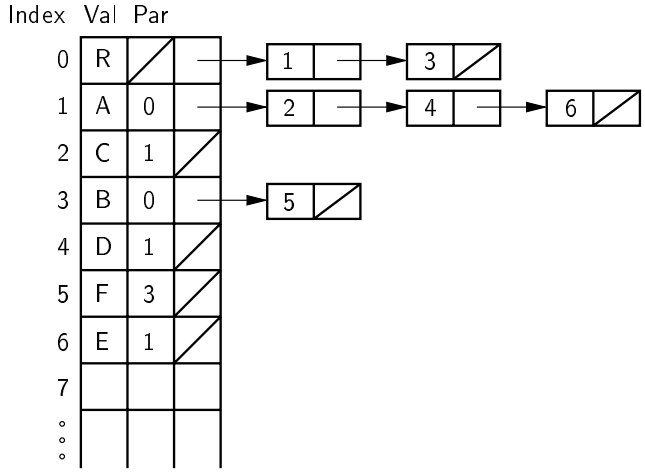


Path Compression Example

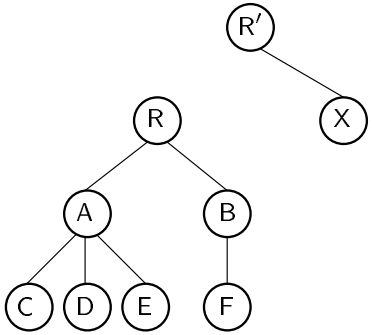


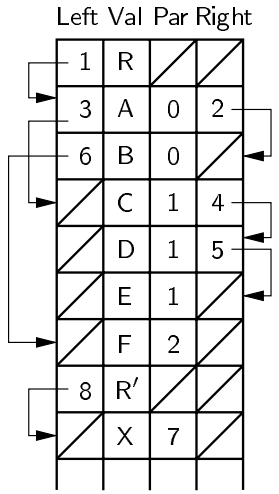

```
GTNode FIND(GTNode curr) {
  if (curr.parent() == null) return curr;  // At root
    return curr.setParent(FIND(curr.parent()));
}
```

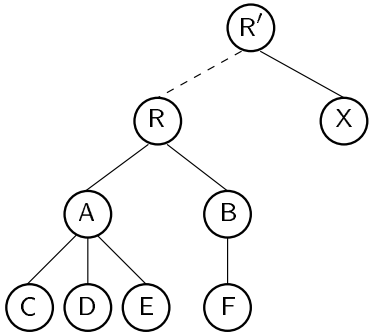

Lists of Children

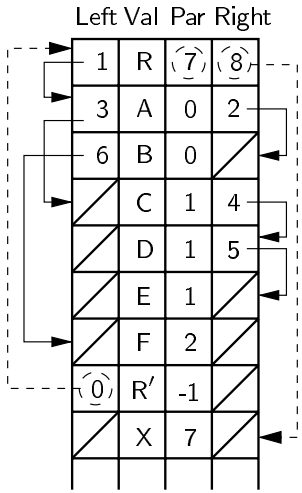


Leftmost Child/Right Sibling

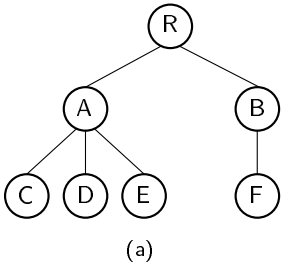




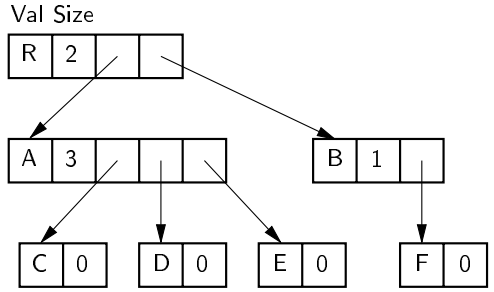




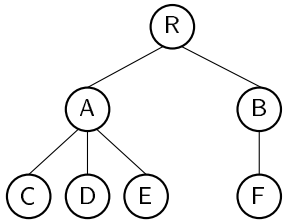
Linked Implementations



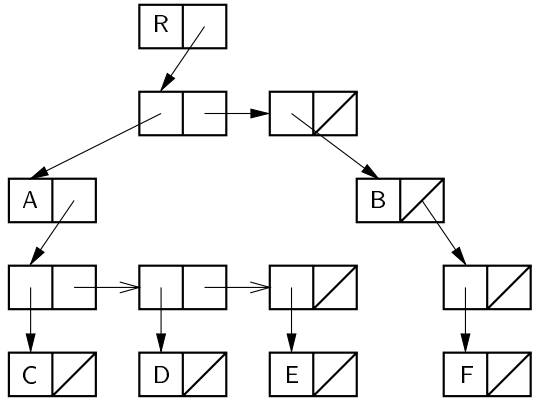
(a)



(b)



(a)



(b)

Sequential Implementations

List node values in the order they would be visited by a preorder traversal.

Saves space, but allows only sequential access.

Need to retain tree structure for reconstruction.

For binary trees: Use symbol to mark `NULL` links.

$$AB/D//CEG///FH//I//$$

Full binary trees: Mark leaf or internal.

$$A'B'/DC'E'G/F'HI$$

General trees: Mark end of each subtree.

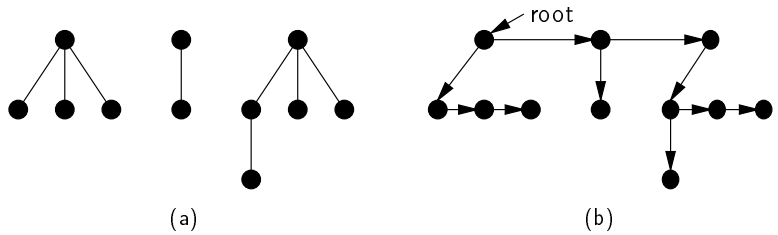
$$RAC)D)E))BF)))$$

Convert to Binary Tree

Left Child/Right Sibling representation essentially stores a binary tree.

Use this process to convert any general tree to a binary tree.

A **forest** is a collection of one or more general trees.



Graphs

A **graph** G = (V, E) consists of a set of **vertices** V, and a set of **edges** E, such that each edge in E is a connection between a pair of vertices in V.

The number of vertices is written |V|, and the number of edges is written |E|.

A sequence of vertices $v_1, v_2, ..., v_n$ forms a **path** of length $n-1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$.
A path is **simple** if all vertices on the path are distinct.

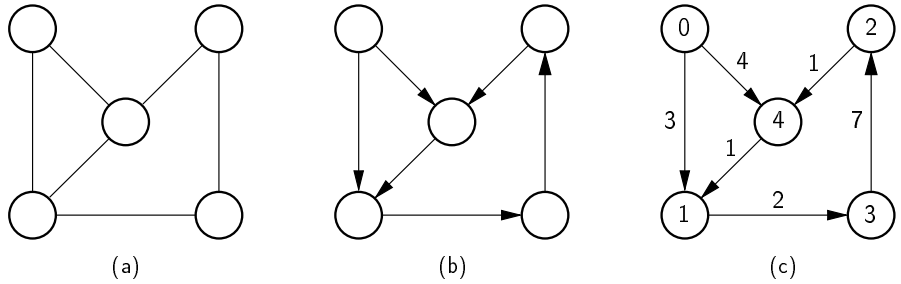
A **cycle** is a path of length 3 or more that connects v_i to itself.
A cycle is **simple** if the path is simple, except for the first and last vertices being the same.

Graph Definitions (Cont)

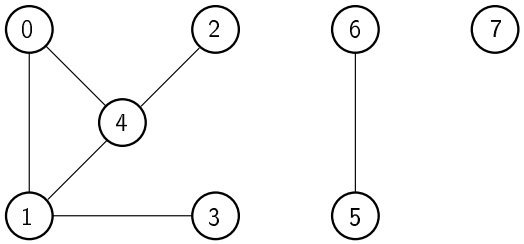
An undirected graph is **connected** if there is at least one path from any vertex to any other.
The maximal connected subgraphs of an undirected graph are called **connected components**.

A graph without cycles is **acyclic**.

A directed graph without cycles is a **directed acyclic graph** or DAG.
A **free tree** is a connected, undirected graph with no simple cycles. Equivalently, a free tree is connected and has |V − 1| edges.

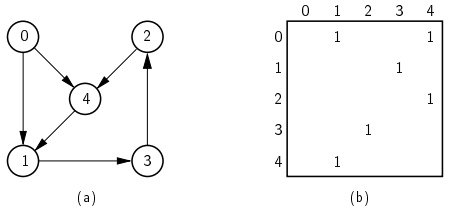


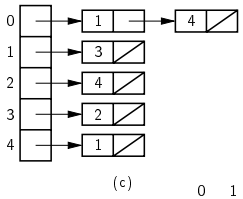
Connected Components

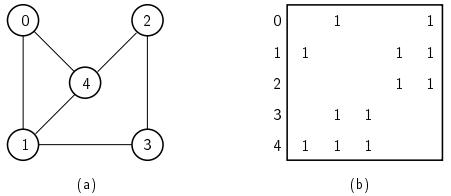


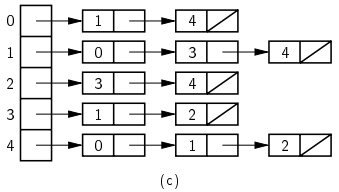
Graph Representations

Adjacency Matrix: $\Theta(|V|^2)$.
Adjacency List: $\Theta(|V| + |E|)$.









Graph Interface

```
interface Graph {                    // Graph class ADT
  public int n();                    // Number vertices
  public int e();                    // Number of edges
  public Edge first(int v);          // Get first edge
  public Edge next(Edge w);          // Get next edge
  public boolean isEdge(Edge w);     // True if edge
  public boolean isEdge(int i, int j); // True if edge
  public int v1(Edge w);             // Where from
  public int v2(Edge w);             // Where to
  public void setEdge(int i, int j, int weight);
  public void setEdge(Edge w, int weight);
  public void delEdge(Edge w);       // Delete edge w
  public void delEdge(int i, int j); // Delete (i, j)
  public int weight(int i, int j);   // Return weight
  public int weight(Edge w);         // Return weight
  public void setMark(int v, int val); // Set Mark
  public int getMark(int v);         // Get Mark
} // interface Graph
```


Implementation: Edge Class

```
interface Edge {   // Interface for graph edges
  public int v1(); // Return the vertex it comes from
  public int v2(); // Return the vertex it goes to
} // interface Edge

// Edge class for Adjacency Matrix graph representation
class Edgem implements Edge {
  private int vert1, vert2; // The vertex indices

  public Edgem(int vt1, int vt2)
    { vert1 = vt1; vert2 = vt2; }
  public int v1() { return vert1; }
  public int v2() { return vert2; }
} // class Edgem
```

Implementation: Adjacency Matrix

```
class Graphm implements Graph { // Adjacency matrix
  private int[][] matrix;       // The edge matrix
  private int numEdge;          // Number of edges
  public int[] Mark;            // The mark array

  public Graphm(int n) {        // Constructor
    Mark = new int[n];
    matrix = new int[n][n];
    numEdge = 0;
  }

  public int n() { return Mark.length; }
  public int e() { return numEdge; }

  public Edge first(int v) { // Get first edge
    for (int i=0; i<Mark.length; i++)
      if (matrix[v][i] != 0)
        return new Edgem(v, i);
    return null;  // No edge for this vertex
  }

  public Edge next(Edge w) { // Get next edge
    if (w == null) return null;
    for (int i=w.v2()+1; i<Mark.length; i++)
      if (matrix[w.v1()][i] != 0)
        return new Edgem(w.v1(), i);
    return null;  // No next edge;
  }
```


Adjacency Matrix (cont)

```
public boolean isEdge(Edge w) { // True if an edge
  if (w == null) return false;
  else return matrix[w.v1()][w.v2()] != 0;
}

public boolean isEdge(int i, int j) // True if edge
  { return matrix[i][j] != 0; }

public int v1(Edge w) {return w.v1();} // Where from
public int v2(Edge w) {return w.v2();} // Where to

public void setEdge(int i, int j, int wt) {
  Assert.notFalse(wt!=0, "Cannot set weight to 0");
  if (matrix[i][j] != 0) numEdge++;
  matrix[i][j] = wt;
}

public void setEdge(Edge w, int weight) // Set weight
  { if (w != null) setEdge(w.v1(), w.v2(), weight); }

public void delEdge(Edge w) {       // Delete edge w
  if (w != null)
    if (matrix[w.v1()][w.v2()] != 0)
      { matrix[w.v1()][w.v2()] = 0; numEdge--; }
}

public void delEdge(int i, int j) { // Delete (i, j)
  if (matrix[i][j] != 0)
    { matrix[i][j] = 0;  numEdge--; }
}
```

Adjacency Matrix (cont 2)

```
  public int weight(int i, int j) {  // Return weight
    if (matrix[i][j] == 0) return Integer.MAX_VALUE;
    else return matrix[i][j];
  }

  public int weight(Edge w) { // Return edge weight
    Assert.notNull(w,"Can't take weight of null edge");
    if (matrix[w.v1()][w.v2()] == 0)
      return Integer.MAX_VALUE;
    else return matrix[w.v1()][w.v2()];
  }

  public void setMark(int v, int val)
    { Mark[v] = val; }

  public int getMark(int v) { return Mark[v]; }
} // class Graphm
```


Implementation: Edge Class

```
// Edge class for Adjacency List graph representation
class Edgel implements Edge {
  private int vert1, vert2;   // Indices of v1, v2
  private Link itself; // Pointer to node in adj list

  public Edgel(int vt1, int vt2, Link it) //Constructor
    { vert1 = vt1;  vert2 = vt2;  itself = it; }

  public int v1() { return vert1; }
  public int v2() { return vert2; }
  Link theLink() { return itself; } // Access adj list
} // class Edgel
```

Implementation: Adjacency List

```
class Graphl implements Graph {// Graph: Adjacency list
  private GraphList[] vertex;  // The vertex list
  private int numEdge;         // Number of edges
  public int[] Mark;           // The mark array

  public Graphl(int n) {       // Constructor
    Mark = new int[n];
    vertex = new GraphList[n];
    for (int i=0; i<n;i++) vertex[i] = new GraphList();
    numEdge = 0;
  }

  public Edge first(int v) { // Get first edge
    vertex[v].setFirst();
    if (vertex[v].currValue() == null) return null;
    return new Edgel(v,
            ((int[])vertex[v].currValue())[0],
                    vertex[v].currLink());
  }

  public boolean isEdge(Edge e) { // True if an edge
    if (e == null) return false;
    vertex[e.v1()].setCurr(((Edgel)e).theLink());
    if (!vertex[e.v1()].isInList()) return false;
    return ((int[])vertex[e.v1()].currValue())[0]
           == e.v2();
  }

  public int v1(Edge e) { return e.v1(); }
  public int v2(Edge e) { return e.v2(); }
```


Adjacency List (cont)

```
public boolean isEdge(int i, int j) { // True if edge
  GraphList temp = vertex[i];
  for (temp.setFirst(); ((temp.currValue()!=null) &&
     (((int[])temp.currValue())[0]<j)); temp.next());
  return (temp.currValue() != null) &&
          (((int[])temp.currValue())[0] == j);
}

public Edge next(Edge e) { // Get next vertex edge
  vertex[e.v1()].setCurr(((Edgel)e).theLink());
  vertex[e.v1()].next();
  if (vertex[e.v1()].currValue()==null) return null;
  return new Edgel(e.v1(),
          ((int[])vertex[e.v1()].currValue())[0],
                   vertex[e.v1()].currLink());
}

public void setEdge(int i, int j, int weight) {
  Assert.notFalse(weight!=0, "Cannot set to 0");
  int[] currEdge = { j, weight };
  if (isEdge(i, j)) vertex[i].setValue(currEdge);
  else                        // Add new edge to graph
    { vertex[i].insert(currEdge); numEdge++; }
}

public void setEdge(Edge w, int weight) // Set weight
  { if (w != null) setEdge(w.v1(), w.v2(), weight); }
```

Adjacency List (cont 2)

```
  public int weight(int i, int j) { // Return weight
    if (isEdge(i, j))
      return ((int[])vertex[i].currValue())[1];
    else return Integer.MAX_VALUE;
  }

  public int weight(Edge e) {    // Return edge weight
    if (isEdge(e))
      return ((int[])vertex[e.v1()].currValue())[1];
    else return Integer.MAX_VALUE;
  }
} // class Graphl
```


Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

Application may require that vertices be visited in some special order based on graph topology.

Example: Artificial Intelligence

- Problem domain consists of many "states."
- Need to get from Start State to Goal State.
- Start and Goal are typically not directly connected.

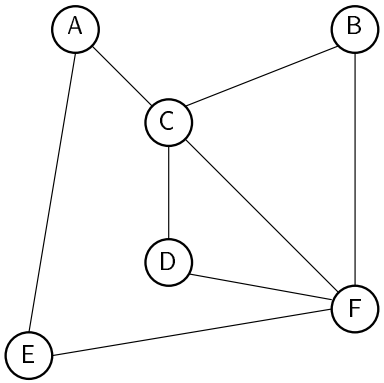
To insure visiting all vertices:

```
void graphTraverse(Graph G) {
  for (v=0; v<G.n(); v++)
    G.setMark(v, UNVISITED);  // Initialize mark bits
  for (v=0; v<G.n(); v++)
    if (G.getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

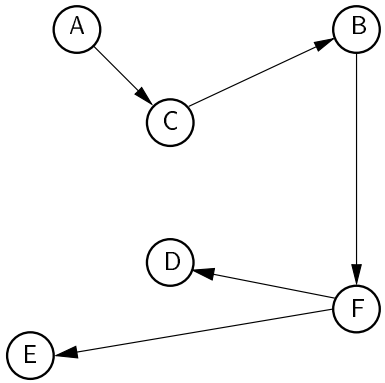
Depth First Search

```
static void DFS(Graph G, int v) { // Depth first search
  PreVisit(G, v);         // Take appropriate action
  G.setMark(v, VISITED);
  for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
    if (G.getMark(G.v2(w)) == UNVISITED)
      DFS(G, G.v2(w));
  PostVisit(G, v);        // Take appropriate action
}
```

Cost: $\Theta(|V| + |E|)$.



(a)



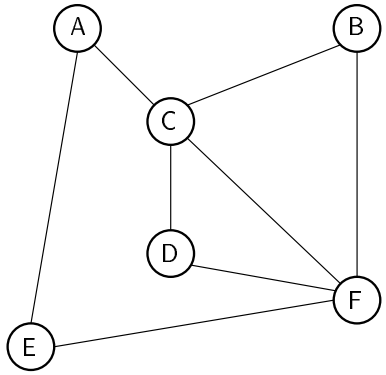
(b)

Breadth First Search

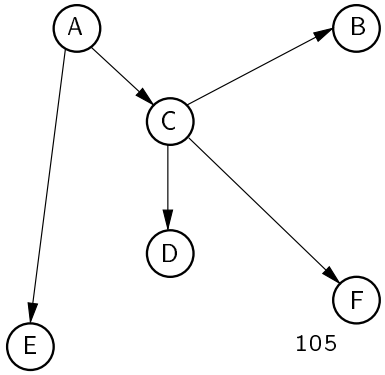
Like DFS, but replace stack with a queue.

Visit the vertex's neighbors before continuing deeper in the tree.

```
static void BFS(Graph G, int start) {
  Queue Q = new AQueue(G.n());        // Use a Queue
  Q.enqueue(new Integer(start));
  G.setMark(start, VISITED);
  while (!Q.isEmpty()) { // Process each vertex on Q
    int v = ((Integer)Q.dequeue()).intValue();
    PreVisit(G, v);          // Take appropriate action
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
      if (G.getMark(G.v2(w)) == UNVISITED) {
        G.setMark(G.v2(w), VISITED);
        Q.enqueue(new Integer(G.v2(w)));
      }
    PostVisit(G, v);         // Take appropriate action
  }
}
```



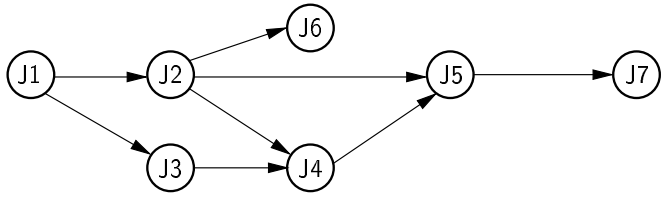
(a)



(b)

Topological Sort

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.




```
static void topsort(Graph G) { // Topo sort: recursive
  for (int i=0; i<G.n(); i++)  // Initialize Mark array
    G.setMark(i, UNVISITED);
  for (int i=0; i<G.n(); i++)  // Process all vertices
    if (G.getMark(i) == UNVISITED)
      tophelp(G, i);           // Call helper function
}

static void tophelp(Graph G, int v) { // Topsort helper
  G.setMark(v, VISITED);
  for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
    if (G.getMark(G.v2(w)) == UNVISITED)
      tophelp(G, G.v2(w));
  printout(v);            // PostVisit for Vertex v
}
```


Queue-based Topological Sort

```
static void topsort(Graph G) { // Topo sort: Queue
  Queue Q = new AQueue(G.n());
  int[] Count = new int[G.n()];
  int v;
  for (v=0; v<G.n(); v++) Count[v] = 0; // Initialize
  for (v=0; v<G.n(); v++)      // Process every edge
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
       Count[G.v2(w)]++;       // Add to v2's count
  for (v=0; v<G.n(); v++)      // Initialize Queue
    if (Count[v] == 0)         // Vertex has no prereqs
      Q.enqueue(new Integer(v));
  while (!Q.isEmpty()) {       // Process the vertices
    v = ((Integer)Q.dequeue()).intValue();
    printout(v);               // PreVisit for Vertex V
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w)) {
      Count[G.v2(w)]--;        // One less prerequisite
      if (Count[G.v2(w)] == 0) // This vertex now free
        Q.enqueue(new Integer(G.v2(w)));
    }
  }
}
```

Shortest Paths Problems

Input: A graph with **weights** or **costs** associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:

- Find the shortest path between two specified vertices.
- Find the shortest path from vertex S to all other vertices.
- Find the shortest path between all pairs of vertices.

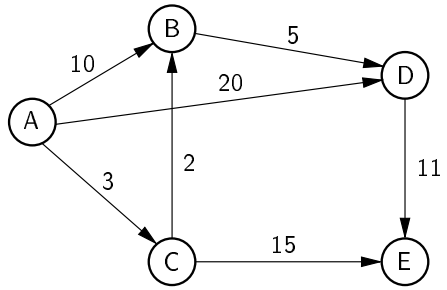
Our algorithms will actually calculate only **distances**.

Shortest Paths Definitions

d(A, B) is the **shortest distance** from vertex A to B.

w(A, B) is the **weight** of the edge connecting A to B.

- If there is no such edge, then w(A, B) $= \infty$.



Single Source Shortest Paths

Given start vertex s, find the shortest path from s to all other vertices.

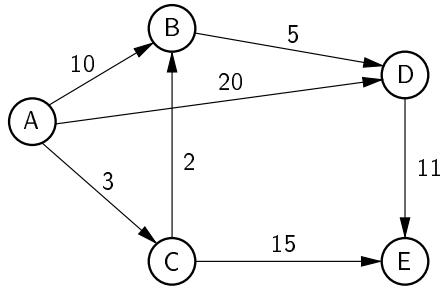
Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.

Problem: Shortest path to a vertex already processed might go through x.

Solution: Process vertices in order of distance from s.

Dijkstra's Algorithm Example

	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18



Dijkstra's Algorithm: Array

```
// Compute shortest path distances from s, store in D
static void Dijkstra(Graph G, int s, int[] D) {
  for (int i=0; i<G.n(); i++)     // Initialize
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) { // Process vertices
    int v = minVertex(G, D); // Get next-closest vertex
    G.setMark(v, VISITED);
    if (D[v] == Integer.MAX_VALUE) return;
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > (D[v] + G.weight(w)))
        D[G.v2(w)] = D[v] + G.weight(w);
  }
}

static int minVertex(Graph G, int[] D) {
  int v = 0;  // Initialize v to any unvisited vertex;
  for (int i=0; i<G.n(); i++)
    if (G.getMark(i) == UNVISITED) { v = i; break; }
  for (int i=0; i<G.n(); i++)  // Find smallest value
    if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
      v = i;
  return v;
}
```

Approach 1: Scan the table on each pass for closest vertex.

Total cost: $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$.

Dijkstra's Algorithm: Priority Queue

```
// Dijkstra's shortest-paths algorithm:
//   priority queue version
static void Dijkstra(Graph G, int s, int[] D) {
  int v;                          // The current vertex
  DijkElem[] E = new DijkElem[G.e()]; // Heap
  E[0] = new DijkElem(s, 0);    // Initialize array
  MinHeap H = new MinHeap(E, 1, G.e()); // Create heap
  for (int i=0; i<G.n(); i++)   // Initialize distances
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) { // For each vertex
    do { v = ((DijkElem)H.removemin()).vertex(); }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (D[v] == Integer.MAX_VALUE) return;
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > (D[v] + G.weight(w))) {
        D[G.v2(w)] = D[v] + G.weight(w);
        H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));
      }
  }
}
```

Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.

Total cost: $\Theta((|V| + |E|) \log |V|)$.

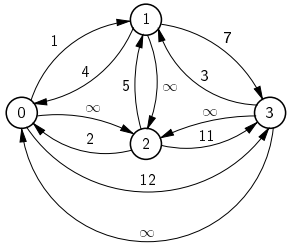
All Pairs Shortest Paths

For every vertex $u, v \in$ V, calculate d(u, v).

Could run Dijkstra's Algorithm V times.

Better is **Floyd's Algorithm**.

Define a **k-path** from u to v to be any path whose intermediate vertices all have indices less than k.



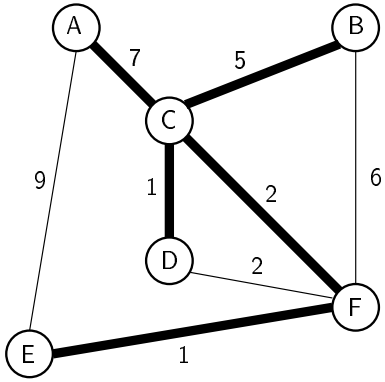
Floyd's Algorithm

```
// Compute all-pairs shortest paths
static void Floyd(Graph G, int[][] D) {
  for (int i=0; i<G.n(); i++) // Initialize D
    for (int j=0; j<G.n(); j++)
      D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute all k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if ((D[i][k] != Integer.MAX_VALUE) &&
            (D[k][j] != Integer.MAX_VALUE) &&
            (D[i][j] > (D[i][k] + D[k][j])))
          D[i][j] = D[i][k] + D[k][j];
}
```

Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:

- Input: An undirected, connected graph G.
- Output: The subgraph of G that 1) has minimum total cost as measured by summing the values for all of the edges in the subset, and 2) keeps the vertices connected.



Prim's MST Algorithm

```
// Compute a minimal-cost spanning tree
static void Prim(Graph G, int s, int[] D) {
  int[] V = new int[G.n()];   // V[i] closest to i
  for (int i=0; i<G.n(); i++) // Initialize
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) { // Process vertices
    int v = minVertex(G, D);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v);
    if (D[v] == Integer.MAX_VALUE) return;
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > G.weight(w)) {
        D[G.v2(w)] = G.weight(w);
        V[G.v2(w)] = v;
      }
  }
}
```

This is an example of a **greedy** algorithm.

Alternative Prim's Implementation

Like Dijkstra's algorithm, we can implement Prim's algorithm with a priority queue.

```
// Prims's MST algorithm: priority queue version
static void Prim(Graph G, int s, int[] D) {
  int v;                    // The current vertex
  int[] V = new int[G.n()]; // V[i] closest to i
  DijkElem[] E = new DijkElem[G.e()]; // Heap
  E[0] = new DijkElem(s, 0);      // Initialize array
  MinHeap H = new MinHeap(E, 1, G.e()); // Create heap
  for (int i=0; i<G.n(); i++) // Initialize dist array
    D[i] = Integer.MAX_VALUE;
  D[s] = 0;
  for (int i=0; i<G.n(); i++) {  // Now, get distances
    do { v = ((DijkElem)H.removemin()).vertex(); }
      while (G.getMark(v) == VISITED);
    G.setMark(v, VISITED);
    if (v != s) AddEdgetoMST(V[v], v); // Add MST edge
    if (D[v] == Integer.MAX_VALUE) return;
    for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
      if (D[G.v2(w)] > G.weight(w)) {  // Update D
        D[G.v2(w)] = G.weight(w);
        V[G.v2(w)] = v;    / Update who it came from
        H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));
      }
  }
}
```


Proof of Prim's MST Algorithm

Theorem 14.1 *Prim's algorithm produces a minimum cost spanning tree.*

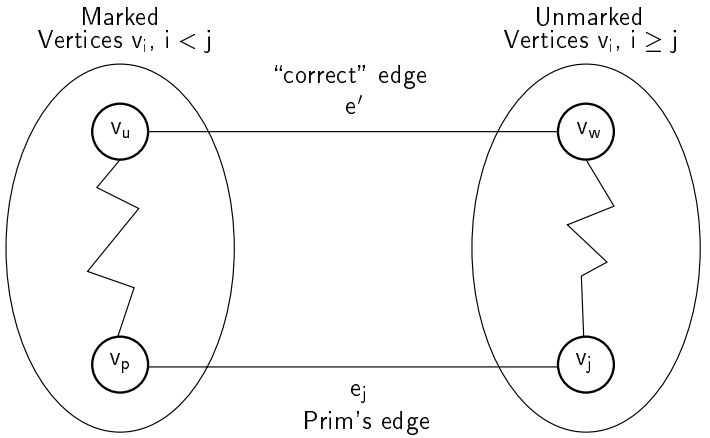
Proof by contradiction:

Order vertices by how they are added to the MST by Prim's algorithm. $v_1, v_2, ..., v_n$.

Let edge e_i connect (v_x, v_{i+1}), $x < i$.

Let e_j be the lowest numbered (first) edge added by the algorithm such that the set of edges selected so far *cannot* be extended to form an MST for G.

Let $V_1 = (v_1, ..., v_j)$. Let $V_2 = (v_{j+1}, ..., v_n)$.



Kruskal's MST Algorithm

```
static void Kruskal(Graph G) { // Kruskal's MST alg
  GenTree A = new GenTree(G.n());
  KruskalElem[] E = new KruskalElem[G.e()];
  int edgecnt = 0; // Count of edges

  for (int i=0; i<G.n(); i++) // Put edges on array
    for (Edge w=G.first(i); G.isEdge(w); w=G.next(w))
      E[edgecnt++] = new KruskalElem(G, w);
  MinHeap H = new MinHeap(E, edgecnt, edgecnt);
  int numMST = G.n();            // Initially n equivs
  for (int i=0; numMST>1; i++) { // Combine equivs
    KruskalElem temp = (KruskalElem)H.removemin();
    Edge w = temp.edge();
    int v = G.v1(w);  int u = G.v2(w);
    if (A.differ(v, u)) {  // If different equivs
      A.UNION(v, u);       //   combine equiv classes
      AddEdgetoMST(G.v1(w), G.v2(w)); // Add MST edge
      numMST--;                  // One less MST
    }
  }
}
```

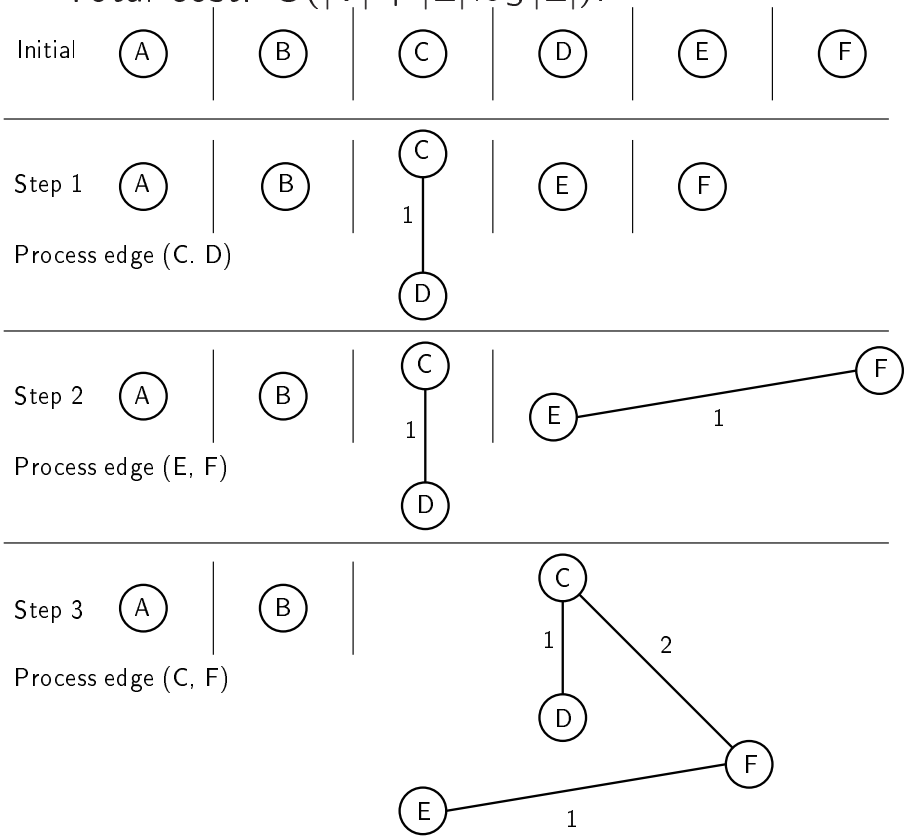
How do we compute function `MSTof(v)`?

Solution: Use Parent Pointer representation to merge equivalance classes.

Kruskal's Algorithm Example

Time dominated by cost for initial edge sort.

Total cost: $\Theta(|V| + |E| \log |E|)$.



Sorting

Each record contains a field called the **key**.
Linear order: comparison.

The Sorting Problem

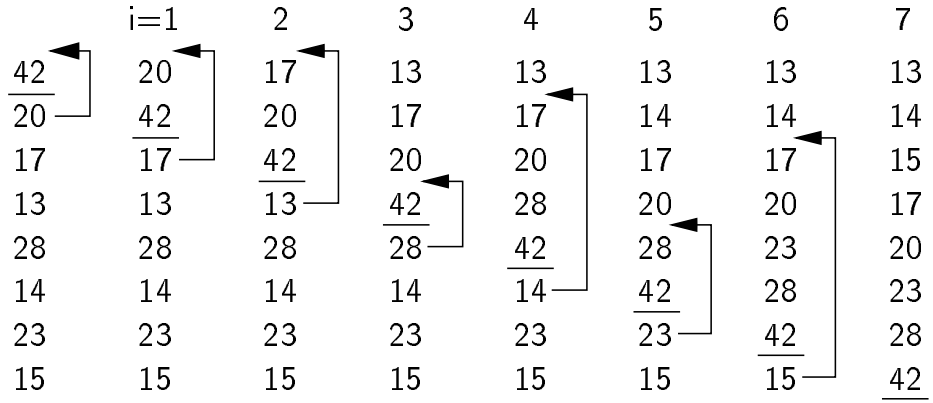
Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

Measures of cost:

- Comparisons
- Swaps

Insertion Sort

```
static void inssort(Elem[] array) {  // Insertion Sort
  for (int i=1; i<array.length; i++) // Insertrecord
    for (int j=i; (j>0) &&
            (array[j].key()<array[j-1].key()); j--)
      DSutil.swap(array, j, j-1);
}
```



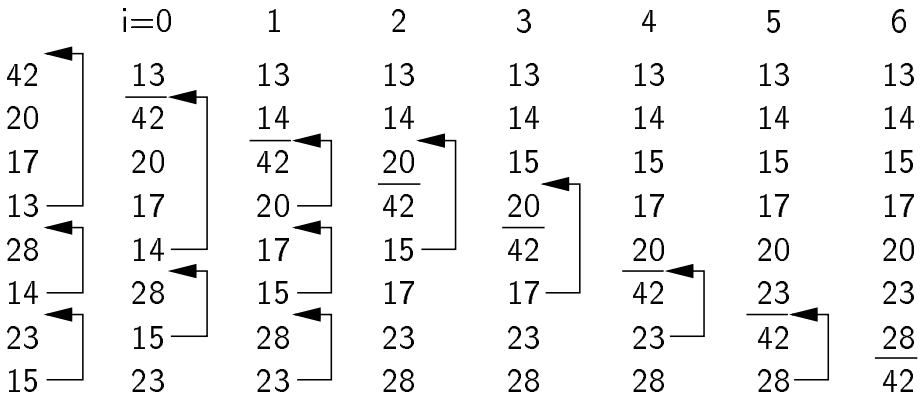
Best Case:

Worst Case:

Average Case:

Bubble Sort

```
static void bubsort(Elem[] array) {    // Bubble Sort
  for (int i=0; i<array.length-1; i++) // Bubble up
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[j-1].key())
        DSutil.swap(array, j, j-1);
}
```



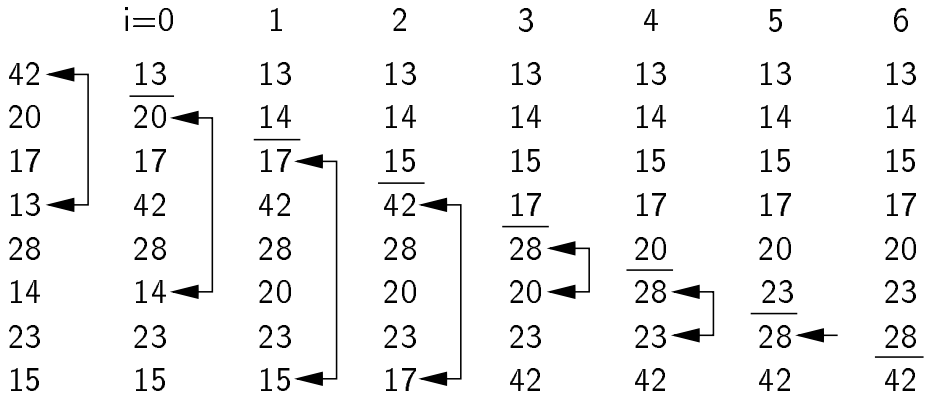
Best Case:

Worst Case:

Average Case:

Selection Sort

```
static void selsort(Elem[] array) {   // Selection Sort
  for (int i=0; i<array.length-1; i++) { // Select i'th
    int lowindex = i;                // Remember its index
    for (int j=array.length-1; j>i; j--) // Find least
      if (array[j].key() < array[lowindex].key())
        lowindex = j;                // Put it in place
    DSutil.swap(array, i, lowindex);
  }
}
```

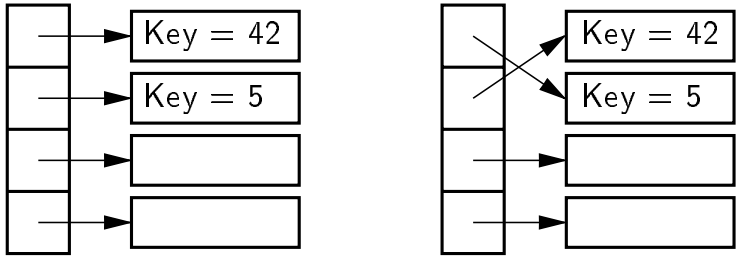


Best Case:

Worst Case:

Average Case:

Pointer Swapping



(a) (b)

Exchange Sorting

Summary

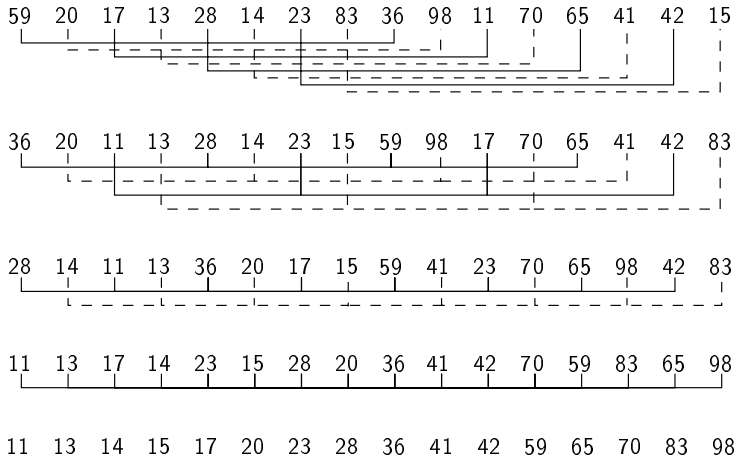
	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

All of these sorts rely on **exchanges** of adjacent records.

What is the average number of exchanges required?

Shellsort

```
static void shellsort(Elem[] array) { // Shellsort
  for (int i=array.length/2; i>2; i/=2)
    for (int j=0; j<i; j++)           // Sort sublists
      inssort2(array, j, i);
  inssort2(array, 0, 1); // Could call regular inssort
}
// Modified version of Ins Sort for varying increments
static void inssort2(Elem[] A, int start, int incr) {
  for (int i=start+incr; i<A.length; i+=incr)
    for (int j=i; (j>=incr)&&
                  (A[j].key()<A[j-incr].key()); j-=incr)
      DSutil.swap(A, j, j-incr);
}
```



$O(n^{1.5})$

Quicksort

Divide and Conquer: divide list into values less than pivot and values greater than pivot.

```
static void qsort(Elem[] array, int i, int j) {
  int pivotindex = findpivot(array, i, j); // Pick piv
  DSutil.swap(array, pivotindex, j); // Stick at end
  // k will be the first position in the right subarray
  int k = partition(array, i-1, j, array[j].key());
  DSutil.swap(array, k, j);    // Put pivot in place
  if ((k-i) > 1) qsort(array, i, k-1);  // Sort left
  if ((j-k) > 1) qsort(array, k+1, j);  // Sort right
}

static int findpivot(Elem[] array, int i, int j)
  { return (i+j)/2; }
```

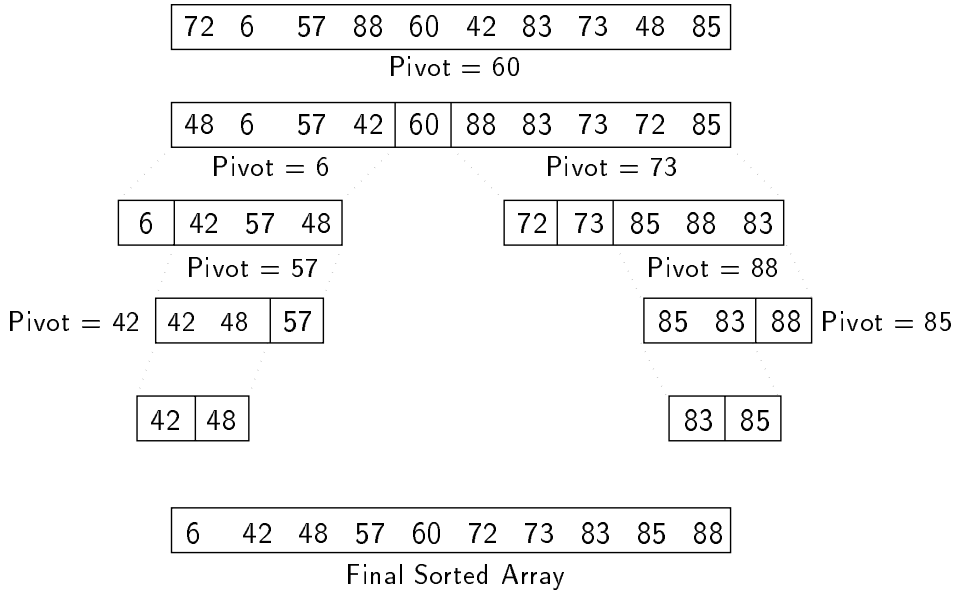
Quicksort Partition

```
static int partition(Elem[] array, int l, int r,
                     int pivot) {
  do {          // Move bounds inward until they meet
    while (array[++l].key() < pivot); // Move it right
    while ((r!=0) && (array[--r].key()>pivot));
    DSutil.swap(array, l, r); // Swap out-of-place vals
  } while (l < r);            // Stop when they cross
  DSutil.swap(array, l, r);   // Reverse wasted swap
  return l; }// Return first pos in right partition
```

Initial	72	6	57	88	85	42	83	73	48	60
	l									r
Pass 1	72	6	57	88	85	42	83	73	48	60
	l								r	
Swap 1	48	6	57	88	85	42	83	73	72	60
	l								r	
Pass 2	48	6	57	88	85	42	83	73	72	60
				l		r				
Swap 2	48	6	57	42	85	88	83	73	72	60
				l		r				
Pass 3	48	6	57	42	85	88	83	73	72	60
				r	l					
Swap 3	48	6	57	85	42	88	83	73	72	60
				r	l					
Reverse Swap	48	6	57	42	85	88	83	73	72	60
				r	l					

The cost for Partition is $\Theta(n)$.

Quicksort Example



Cost for Quicksort

Best Case: Always partition in half.

Worst Case: Bad partition.

Average Case:

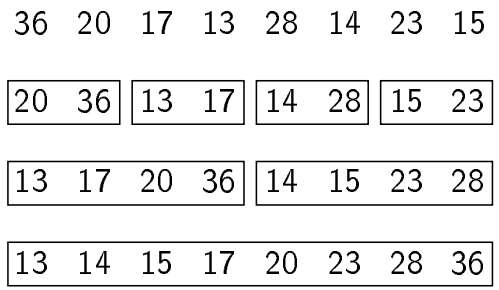
$$\begin{aligned} T(n) &= n + 1 + \frac{1}{n-1} \sum_{k=1}^{n-1} (T(k) + T(n-k)) \\ &= \Theta(n \log n) \end{aligned}$$

Optimizations for Quicksort:

- Better pivot.
- Use better algorithm for small sublists.
- Eliminate recursion.

Mergesort

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List l1 = half of the items from inlist;
  List l2 = other half of the items from inlist;
  return merge(mergesort(l1), mergesort(l2));
}
```



Mergesort Implementation

Mergesort is tricky to implement.

```
static void mergesort(Elem[] array, Elem[] temp,
                      int l, int r) {
  int mid = (l+r)/2;                // Select midpoint
  if (l == r) return;               // One element list
  mergesort(array, temp, l, mid);   // Ssort first half
  mergesort(array, temp, mid+1, r); // Sort second half
  for (int i=l; i<=r; i++)          // Copy subarray
    temp[i] = array[i];
  // Do the merge operation back to array
  int i1 = l; int i2 = mid + 1;
  for (int curr=l; curr<=r; curr++) {
    if (i1 == mid+1)  // Left sublist exhausted
      array[curr] = temp[i2++];
    else if (i2 > r)  // Right sublist exhausted
      array[curr] = temp[i1++];
    else if (temp[i1].key() < temp[i2].key())
      array[curr] = temp[i1++]; // Get smaller val
    else array[curr] = temp[i2++];
  }
}
```

Mergesort cost:

Mergesort is good for sorting linked lists.

Optimized Mergesort

```
static void mergesort(Elem[] array, Elem[] temp,
                      int l, int r) {
  int i, j, k, mid = (l+r)/2;  // Select the midpoint
  if (l == r) return;          // List has one element
  if ((mid-l) >= THRESHOLD)
    mergesort(array, temp, l, mid);
  else inssort(array, l, mid-l+1);
  if ((r-mid) > THRESHOLD)
    mergesort(array, temp, mid+1, r);
  else inssort(array, mid+1, r-mid);
  // Do the merge operation.  Copy 2 halves to temp.
  for (i=l; i<=mid; i++) temp[i] = array[i];
  for (j=1; j<=r-mid; j++) temp[r-j+1] = array[j+mid];
  // Merge sublists back to array
  int a = temp[l].key(); int b = temp[r].key();
  for (i=l,j=r,k=l; k<=r; k++)
    if (a < b)
      { array[k] = temp[i++]; a = temp[i].key(); }
    else { array[k] = temp[j--]; b = temp[j].key(); }
}
```

Heapsort

Heapsort uses a max-heap.

```
static void heapsort(Elem[] array) {  // Heapsort
  MaxHeap H = new MaxHeap(array, array.length,
                          array.length);
  for (int i=0; i<array.length; i++)  // Now sort
    H.removemax();   // Put max value at end of heap
}
```

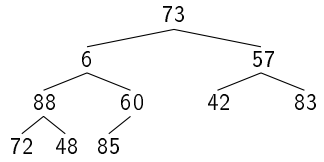
Cost of Heapsort:

Cost of finding k largest elements:

Heapsort Example

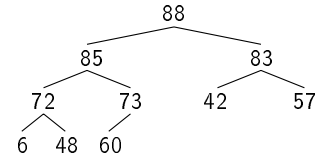
Original Numbers

73	6	57	88	60	42	83	72	48	85



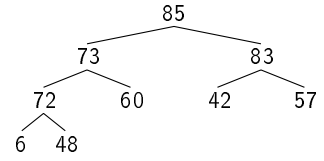
Build Heap

88	85	83	72	73	42	57	6	48	60



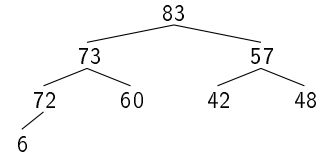
Remove 88

85	73	83	72	60	42	57	6	48	88



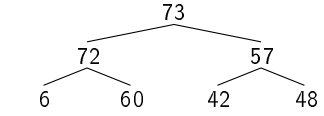
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Binsort

A simple, efficient sort:

```
for (i=0; i<n; i++)
  B[A[i].key()] = A[i];
```

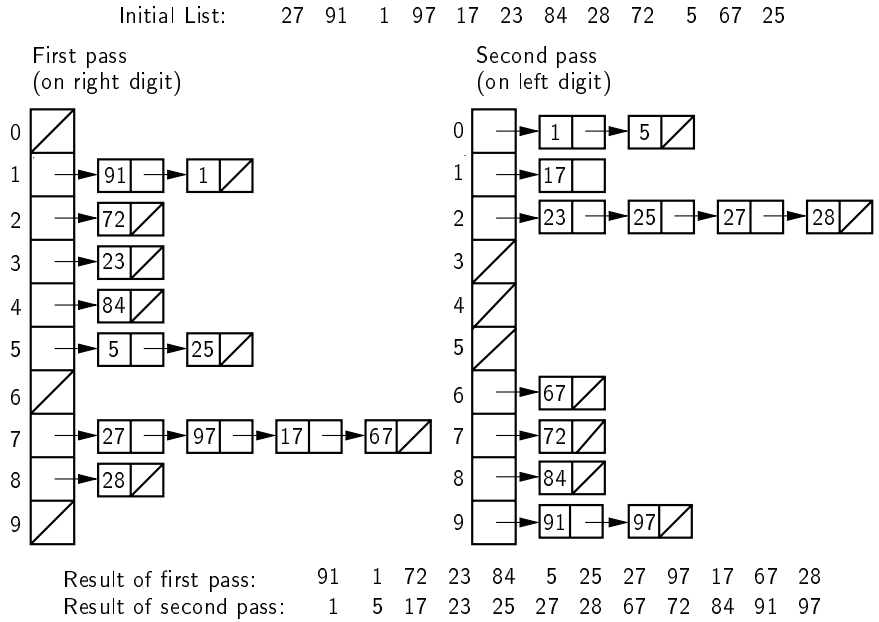
Ways to generalize:

- Make each bin the head of a list.
- Allow more keys than records.

```
void binsort(ELEM *A, int n) {
  list B[MaxKeyValue];
  for (i=0; i<n; i++) B[A[i].key()].append(A[i]);
  for (i=0; i<MaxKeyValue; i++)
    for (B[i].first(); B[i].isInList(); B[i].next())
      output(B[i].currValue());
}
```

Cost:

Radix Sort



Cost of Radix Sort

```
static void radix(Elem[] A, Elem[] B,
                  int k, int r, int[] count) {
  // Count[i] stores number of records in bin[i]
  int i, j, rtok;

  for (i=0, rtok=1; i<k; i++, rtok*=r) {// For k digits
    for (j=0; j<r; j++) count[j] = 0;   // Initialize

    // Count number of recs for each bin on this pass
    for (j=0; j<A.length; j++)
      count[(A[j].key()/rtok)%r]++;

    // Index B: count[j] is index for last slot of j.
    for (j=1; j<r; j++) count[j] = count[j-1]+count[j];

    // Put recs in bins, work from bottom of each bin.
    // Since bins fill from bottom, j counts downwards
    for (j=A.length-1; j>=0; j--)
      B[--count[(A[j].key()/rtok)%r]] = A[j];

    for (j=0; j<A.length; j++) A[j] = B[j]; // Copy
  }
}
```

Cost: $\Theta(nk + rk)$.

How do n, k and r relate?

Radix Sort Example

Initial Input: Array A

27	91	1	97	17	23	84	28	72	5	67	25

First pass values for Count.
rtok = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

End of Pass 1: Array A.

91	1	72	23	84	5	25	27	97	17	67	28

Second pass values for Count.
rtok = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97

Empirical Comparison

Algorithm	10	100	1000	10,000
Insert. Sort	.10	9.5	957.9	98,086
Bubble Sort	.13	14.3	1470.3	157,230
Select. Sort	.11	9.9	1018.9	104,897
Shellsort	.09	2.5	45.6	829
Quicksort	.15	1.8	23.6	291
Quicksort/O	.10	1.6	20.9	274
Mergesort	.12	2.4	36.8	505
Mergesort/O	.08	1.8	28.0	390
Heapsort	–	50.0	60.0	880
Radix Sort/1	.87	8.6	89.5	939
Radix Sort/4	.23	2.3	22.5	236
Radix Sort/8	.19	1.2	11.5	115

Algorithm	10	100	1000	10,000
Insert. Sort	.66	65.9	6423	661,711
Bubble Sort	.90	85.5	8447	1,068,268
Select. Sort	.73	67.4	6678	668,056
Shellsort	.62	18.5	321	5,593
Quicksort	.92	12.7	169	1,836
Quicksort/O	.65	10.7	141	1,781
Mergesort	.76	16.8	234	3,231
Mergesort/O	.53	11.8	189	2,649
Heapsort	–	41.0	565	7,973
Radix Sort/1	7.40	67.4	679	6,895
Radix Sort/4	2.10	18.7	160	1,678
Radix Sort/8	4.10	11.5	97	808

Sorting Lower Bound

Want to prove a lower bound for *all possible* sorting algorithms.

Sorting is $O(n \log n)$.

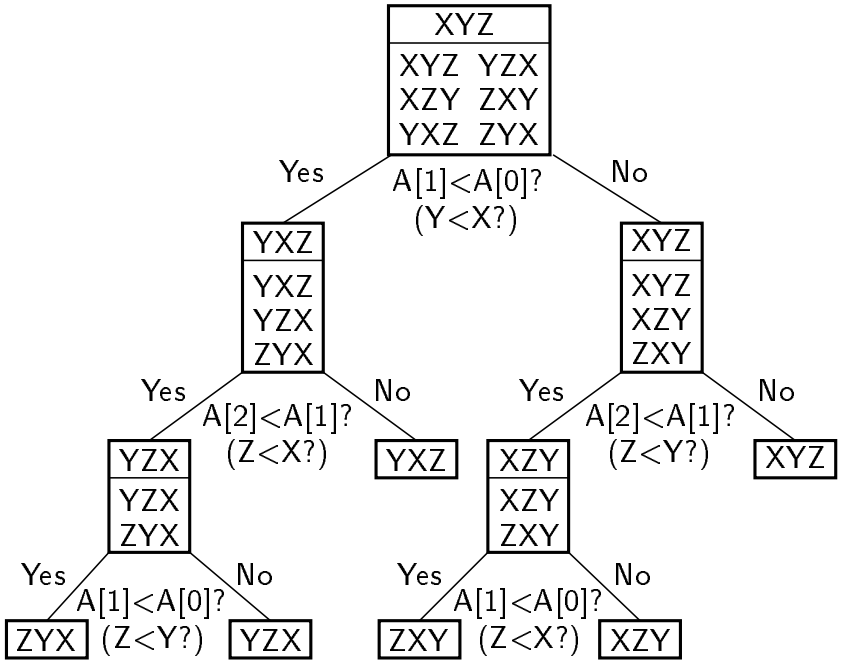
Sorting I/O takes $\Omega(n)$ time.

Will now prove $\Omega(n \log n)$ lower bound.

Form of proof:

- Comparison based sorting can be modeled by a binary tree.
- The tree must have $\Omega(n!)$ leaves.
- The tree must be $\Omega(n \log n)$ levels deep.

Decision Trees



There are $n!$ permutations, and at least 1 node for each permutation.

A tree with n nodes has at least $\log n$ levels.

Where is the worst case in the decision tree?

$\log n! = \Omega(n \log n)$.

Primary vs. Secondary Storage

Primary Storage: Main memory (RAM)

Secondary Storage: Peripheral devices

- Disk Drives
- Tape Drives

Medium	Price	Price per Mbyte
32MB RAM	$225	$7.00/MB
1.4MB floppy disk	$.50	$0.36/MB
2.1GB disk drive	$210	$0.10/MB
1GB JAZ cassette	$100	$0.10/MB
2GB cartridge tape	$20	$0.01/MB

RAM is usually **volatile**.

RAM is about 1/4 million times faster than disk.

Golden Rule of File Processing

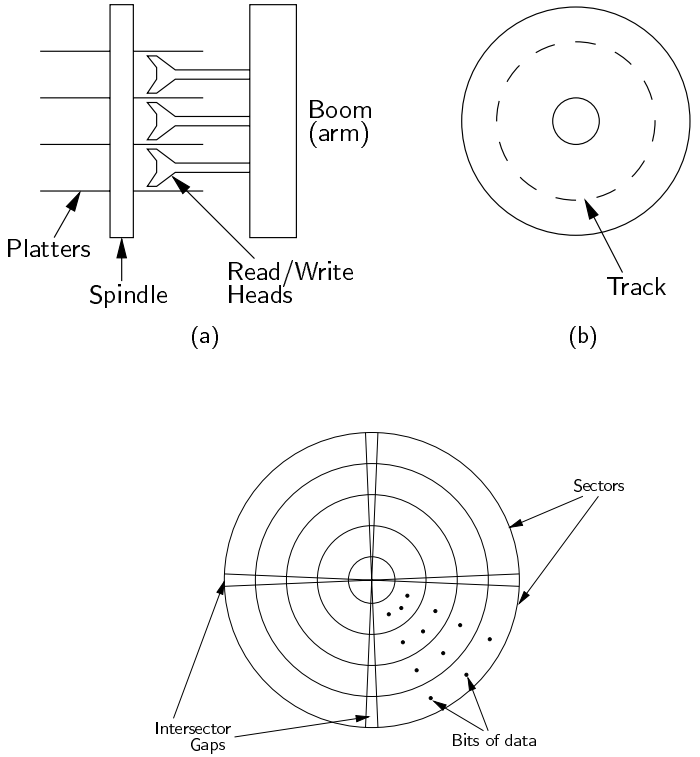
Minimize the number of disk accesses!

1. Arrange information so that you get what you want with few disk accesses.
2. Arrange information so minimize future disk accesses.

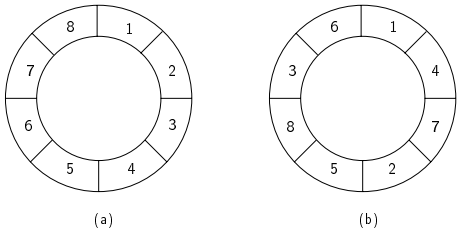
An organization for data on disk is often called a **file structure**.

Disk based space/time tradeoff: Compress information to save processing time by reducing disk accesses.

Disk Drives



Sectors



A **sector** is the basic unit of I/O.

Interleaving factor: Physical distance between logically adjacent sectors on a track, to allow for processing of sector data.

Locality of Reference: If a record is read from disk, the next request is likely to come from near the same place in the file.

Cluster: Smallest unit of file allocation – several sectors.

Extent: A group of physically contiguous clusters.

Internal Fragmentation: Wasted space within a sector if record size does not match sector size, or wasted space within a cluster if file size is not a multiple of cluster size.

Factors affecting disk access time

1. **Seek time**: time for I/O head to reach desired track. Largely determined by distance between I/O head and desired track.

   Seek time: $f(n) = t * n + s$ where t is time to traverse one track and s is startup time for the I/O head.

2. **Rotational delay** or **latency**: time for data to rotate to I/O head position.

   At 3600 RPM, one half rotation of the disk: 16.7/2 msec = 8.3 msec.

3. **Transfer time**: time for data to move under the I/O head.

   Number of sectors read / Number of sectors per track * 16.7 msec

Disk Access Cost Example

675 Mbyte disk drive

- 15 platters $\Rightarrow$ 45 Mbyte/platter
- 612 tracks/platter
- 150 sectors/track $\Rightarrow$ 512 bytes/sector
- 8 sectors/cluster (4K bytes/cluster) $\Rightarrow$ 18 clusters/track
- Interleaving factor of 3 $\Rightarrow$ 3 revolutions to read one track (50.1 msec)

How long to read a file of 128 Kbytes divided into 256 records of 512 bytes?

Number of Clusters:

If file fills minimum number of tracks:

150 sectors of one track, 106 of the next

Total time:

$$612/3 * 0.08 + 3 + 3.5 * 16.7 + 0.08 + 3 + 3.5 * 16.7 = 139.3 \text{ msec.}$$

If clusters are spread randomly across disk:

$$32 * (\frac{612}{3} * 0.08 + 3 + 16.7/2 + \frac{24}{150} * 16.7) = 32 * 30.3 = 969.6 \text{ msec.}$$

Magnetic Tape

Example: 9 track tape at 6250 bytes per inch (bpi).

At 2400 feet, this yields 170 Mbytes for $20, or $0.12/Mbyte.

Workstation/PC cartridge tape is similar.

Magnetic tape requires **sequential** access.

Magnetic tape has two speeds:

- High speed for "skipping."
- Low speed for "reading."

Interblock Gap is space required for I/O head to recognize beginning of a record at high speed. This is a significant amount of space compared to typical record size.

Blocking Factor: Number of records in a block between gaps.

Buffers

Read time for one track:

$$612/3 * 0.08 + 3 + 3.5 * 16.7 = 77.8 \text{ msec.}$$

Read time for one sector:

$612/3*0.08+3+16.7/2+16.7/150 = 27.8$ msec.

Read time for one byte:

$$612/3 * 0.08 + 3 + 16.7/2 = 27.7 \text{ msec.}$$

Nearly all disk drives read/write one sector at every I/O access.

- Also called a **page**.

The information in a sector is stored in a **buffer** or **cache**.

If next I/O access is to same buffer, then no need to go to disk.

There are usually one or more input buffers and one or more output buffers.

Buffer Pools

A series of buffers used by an application to cache disk data is called a **buffer pool**.

Virtual memory uses a buffer pool to imitate greater RAM memory by actually storing information on disk and "swapping" between disk and RAM.

Organization for buffer pools: which one to use next?

- First-in, First-out: Use the first one on the queue.
- Least Frequently Used (LFU): Count buffer accesses, pick the least used.
- Least Recently Used (LRU):
  Keep buffers on linked list.
  When a buffer is accessed, bring to front.
  Reuse the one at the end.

Double Buffering: Read data from disk for next buffer while CPU is processing previous buffer.

Programmer's View of Files

Logical view of files:

- An array of bytes.
- A **file pointer** marks the current position.

Three fundamental operations:

- Read bytes from current position (move file pointer).
- Write bytes to current position (move file pointer).
- Set file pointer to specified byte position.

Java File Functions

```
RandomAccessFile(String name, String mode)

close()

read(byte[] b)

write(byte[] b)

seek(long pos)
```

External Sorting

Problem: Sorting data sets too large to fit in main memory.

- Assume data stored on disk drive.

To sort, portions of the data must be brought into main memory, processed, and returned to disk.

An external sort should minimize disk accesses.

Model of External Computation

Secondary memory is divided into equal-sized **blocks** (512, 2048, 4096 or 8192 bytes are typical sizes).

The basic I/O operation transfers the contents of one disk block to/from main memory.

Under certain circumstances, reading blocks of a file in sequential order is more efficient. (When?)

Typically, the time to perform a single block I/O operation is sufficient to Quicksort the contents of the block.

Thus, our primary goal is to minimize the number fo block I/O operations.

Most workstations today must do all sorting on a single disk drive.

Key Sorting

Often records are large while keys are small.

- Ex: Payroll entries keyed on ID number.

Approach 1: Read in entire records, sort them, then write them out again.

Approach 2: Read only the key values, store with each key the location on disk of its associated record.

If necessary, after the keys are sorted the records can be read and re-written in sorted order.

External Sort: Simple Mergesort

Quicksort requires random access to the entire set of records.

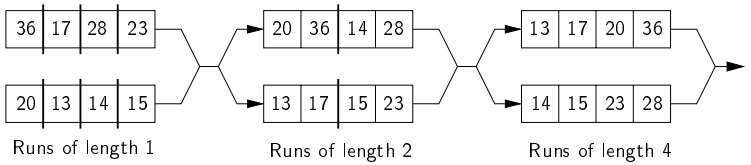
Better: Modified Mergesort algorithm

- Process n elements in $\Theta(\log n)$ passes.

1. Split the file into two files.
2. Read in a block from each file.
3. Take first record from each block, output them in sorted order.
4. Take next record from each block, output them to a *second* file in sorted order.
5. Repeat until finished, alternating between output files. Read new input blocks as needed.
6. Repeat steps 2-5, except this time the input files have groups of two sorted records that are merged together.
7. Each pass through the files provides larger and larger groups of sorted records.

A group of sorted records is called a **run**.

Problems with Simple Mergesort



Is each pass through input and output files sequential?

What happens if all work is done on a single disk drive?

How can we reduce the number of Mergesort passes?

In general, external sorting consists of two phases:

1. Break the file into initial runs.
2. Merge the runs together into a single sorted run.

Breaking a file into runs

General approach:

- Read as much of the file into memory as possible.
- Perform and in-memory sort.
- Output this group of records as a single run.

Replacement Selection

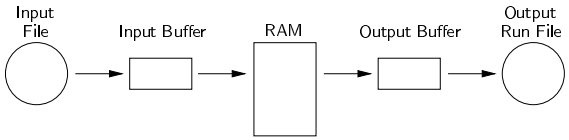
1. Break available memory into an array for the heap, an input buffer and an output buffer.
2. Fill the array from disk.
3. Make a min-heap.
4. Send the smallest value (root) to the output buffer.
5. If the next key in the file is greater than the last value output, then

   Replace the root with this key.

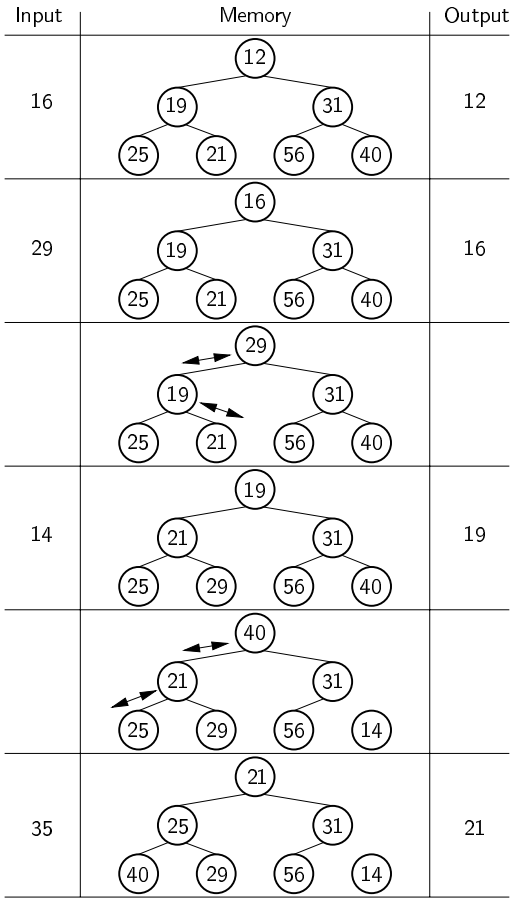
   else

   Replace the root with the last key in the array.

   Add the next record in the file to a new heap (actually, stick it at the end of the array).



Example of Replacement Selection



Benefit from Replacement Selection

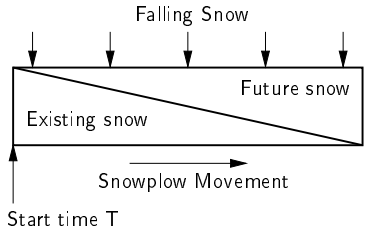
Use double buffering to overlap input, processing and output.

How many disk drives for greatest advantage?

Snowplow argument:

- A snowplow moves around a circular track onto which snow falls at a steady rate.
- At any instant, there is a certain amount of snow S on the track. Some falling snow comes in front of the plow, some behind.
- During the next revolution of the snowplow, all of this is removed, plus 1/2 of what falls during that revolution.
- Thus, the plow removes $2S$ amount of snow.

Is this always true?



Simple Mergesort may not be Best

Simple Mergesort: Place the runs into two files.

- Merge the first two runs to output file, then next two runs, etc.

This process is repeated until only one run remains.

- How many passes for r initial runs?

Is there benefit from sequential reading?

Is working memory well used?

Need a way to reduce the number of passes.

Multiway Merge

With replacement selection, each initial run is several blocks long.

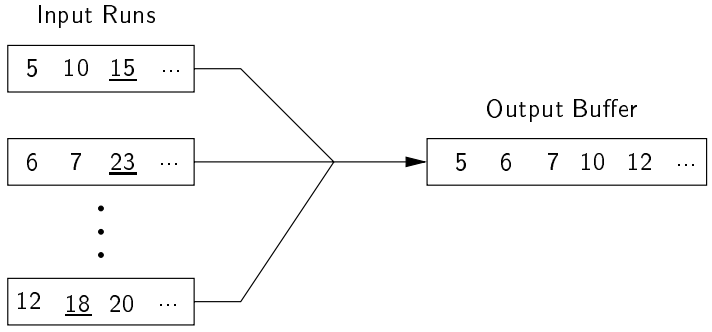
Assume that each run is placed in a separate disk file.

We could then read the first block from each file into memory and perform an r-way merge.

When a buffer becomes empty, read a block from the appropriate run file.

Each record is read only *once* from disk during the merge process.

In practice, use only one file and seek to appropriate block.



Limits to Single Pass Multiway Merge

Assume working memory is b blocks in size.

How many runs can be processed at one time?

The runs are $2b$ blocks long (on average).

How big a file can be merged in one pass?

Larger files will need more passes – but the run size grows quickly!

This approach trades $\Theta(\log b)$ (possibly) sequential passes for a single or a very few random (block) access passes.

General Principals of External Sorting

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing.
- If possible, use additional disk drives for more overlapping of processing with I/O, and allow for more sequential file processing.

Search

Given: Distinct keys k_1, k_2, ... k_n and collection T of n records of the form

$$(k_1, I_1), (k_2, I_2), ..., (k_n, I_n)$$

where I_j is information associated with key k_j for $1 \leq j \leq n$.

Search Problem: For key value K, locate the record (k_j, I_j) in T such that $k_j = K$.

Searching is a systematic method for locating the record (or records) with key value $k_j = K$.

A **successful** search is one in which a record with key $k_j = K$ is found.

An **unsuccessful** search is one in which no record with $k_j = K$ is found (and presumably no such record exists).

Approaches to Search

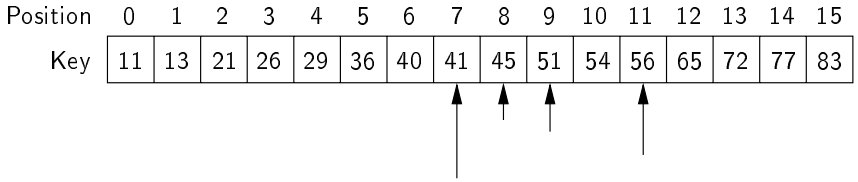
1. Sequential and list methods (lists, tables, arrays).
2. Direct access by key value (hashing).
3. Tree indexing methods.

Searching Ordered Arrays

Sequential Search

Binary Search

```
static int binary(int K, int[] array,
                  int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;   // l and r are beyond array bounds
  while (l+1 != r) { // Stop when l and r meet
    int i = (l+r)/2; // Look at middle of subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```



Dictionary Search

Lists Ordered by Frequency

Order lists by (expected) frequency of occurrance.

- Perform sequential search.

Cost to access first record: 1
Cost to access second record: 2

Expected search cost:

$$\overline{C}_n = 1p_1 + 2p_2 + ... + np_n$$

Example: all records have equal frequency

$$\overline{C}_n = \sum_{i=1}^{n} i/n = (n+1)/2.$$

Example: Exponential frequency

$$p_i = \begin{cases} 1/2^i & \text{if } 1 \leq i \leq n-1 \\ 1/2^{n-1} & \text{if } i = n \end{cases}$$

$$\overline{C}_n \approx \sum_{i=1}^{n} (i/2^i) \approx 2.$$

Zipf Distributions

Applications:

- Distribution for frequency of word usage in natural languages.
- Distribution for populations of cities, etc.

Definition: Zipf frequency for item i in the distribution for n records as $1/i\mathcal{H}_n$.

$$\overline{C}_n = \sum_{i=1}^{n} i/i\mathcal{H}_n = n/\mathcal{H}_n \approx n/\log_e n$$

80/20 rule: 80% of the accesses are to 20% of the records.

For distributions following the 80/20 rule,

$$\overline{C}_n \approx 0.122n.$$

Self-Organizing Lists

Self-organizing lists modify the order of records within the list based on the actual pattern of record access.

Self-organizing lists use a rule called a **heuristic** for deciding how to to reorder the list. These heuristics are similar to the rules for managing buffer pools.

- Order by actual historical frequency of access. (Similar to LFU buffer pool replacement stratagy.)
- When a record is found, swap it with the first record on list.
- **Move-to-Front**: When a record is found, move it to the front of the list.
- **Transpose**: When a record is found, swap it with the record ahead of it.

Example of Self-Organizing Tables

Application: Text compression.

Keep a table of words already seen, organized via Move-to-Front Heuristic.

If a word not yet seen, send the word.
Otherwise, send the (current) index in the table.

The car on the left hit the car I left.

The car on 3 left hit 3 5 I 5.

This is similar in spirit to Ziv-Lempel coding.

Searching in Sets

For dense sets (small range, many elements in set):

Can use logical bit operators.

Example: To find all primes that are odd numbers, compute

0011010100010100 & 0101010101010101

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

Hashing

Hashing: The process of mapping a key value to a position in a table.

A **hash function** maps key values to positions It is denoted by **h**.

A **hash table** is an array that holds the records. It is denoted by T.

The hash table has M slots, indexed from 0 to $M-1$.

For any value K in the key range and some hash function **h**,
h$(K) = i, 0 \leq i < M$, such that $T[i]$.key() $= K$.

Hashing (continued)

Hashing is appropriate only for sets (no duplicates).

Good for both in-memory and disk based applications.

Answers the question "What record, if any, has key value K?"

Example: Store the n records with keys in range 0 to $n-1$.

- Store the record with key i in slot i.
- Use hash function $\mathbf{h}(K) = K$.

Collisions

More reasonable example:

- Store about 1000 records with keys in range 0 to 16,383.
- Impractical to keep a hash table with 16,384 slots.
- We must devise a hash function to map the key range to a smaller table.

Given: hash function **h** and keys k_1 and k_2.
β is a slot in the hash table.
If $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$, then k_1 and k_2 have a **collision** at β under **h**.

Search for the record with key K:

1. Compute the table location $\mathbf{h}(K)$.
2. Starting with slot $\mathbf{h}(K)$, locate the record containing key K using (if necessary) a **collision resolution policy**.

Collisions are inevitable in most applications.

- Example: 23 people are likely to share a birthday.

Hash Functions

A hash function MUST return a value within the hash table range.

To be practical, a hash function SHOULD evenly distribute the records stored among the hash table slots.

Ideally, the hash function should distribute records with equal probability to all hash table slots. In practice, success depends on the distribution of the actual records stored.

If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.

If we have knowlege of the incoming distribution, use a distribution-dependant hash function.

Example Hash Functions

```
static int h(int x) {
  return(x % 16);
}
```

This function is entirely dependant on the lower 4 bits of the key.

Mid-square method: square the key value, take the middle r bits from the result for a hash table of 2^r slots.

Sum the ASCII values of the letters and take results modulo M.

```
static int h(String x, int M) {
  int i, sum;
  for (sum=0, i=0; i<x.length(); i++)
    sum += (int)x.charAt(i);
  return(sum % M);
}
```

ELF Hash

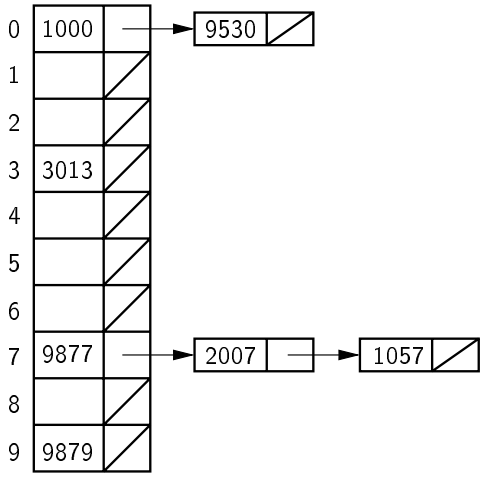
From Executable and Linking Format (ELF), UNIX System V Release 4.

```
static long ELFhash(String key, int M) {
  long h = 0;
  for (int i=0; i<key.length(); i++) {
    h = (h << 4) + (int) key.charAt(i);
    long g = h & 0xF0000000L;
    if (g != 0) h ^= g >>> 24;
    h &= ~g;
  }
  return h % M;
}
```


Open Hashing

What to do when collisions occur?

Open hashing treats each hash table slot as a bin.



Bucket Hashing

Divide the hash table slots into buckets.

- Example: 8 slots/bucket.

Include an overflow bucket.

Records hash to the first slot of the bucket, and fill bucket. Go to overflow if necessary.

When searching, first check the proper bucket. Then check the overflow.

Closed Hashing

Closed hashing stores all records directly in the hash table.

Each record i has a **home position** $\mathbf{h}(k_i)$.

If i is to be inserted and another record already occupies i's home position, then another slot must be found to store i.

The new slot is found by a **collision resolution policy**.

Search must follow the same policy to find records not in their home slots.

Collision Resolution

During insertion, the goal of collision resolution is to find a free slot in the table.

Probe Sequence: the series of slots visited during insert/search by following a collision resolution policy.

Let $\beta_0 = \mathbf{h}(K)$. Let $(\beta_0, \beta_1, ...)$ be the series of slots making up the probe sequence.

```
void hashInsert(ELEM R) { // Insert R into hash table T
  int home;                  // Home position for R
  int pos = home = h(key(R));// Initial pos on sequence
  for (int i=1; key(T[pos]) != EMPTY; i++) {
    pos = (home + p(key(R), i)) % M; // Next slot
    if (key(T[pos]) == key(R)) ERROR; // No duplicates
  }
  T[pos] = R;                 // Insert R
}

ELEM hashSearch(KEY K) { // Search for record w/ key K
  int home;              // Home position for K
  int pos = home = h(K); // Initial pos on sequence
  for (int i = 1; (key(T[pos]) != K) &&
                  (key(T[pos]) != EMPTY); i++)
    pos = (home + p(K, i)) % M; // Next pos on sequence
  if (key(T[pos] == K)) return T[pos];  // Found it
  else return UNSUCCESSFUL;     // K not in hash table
}
```


Linear Probing

Use the probe function

```
int p(int K, int i) { return i; }
```

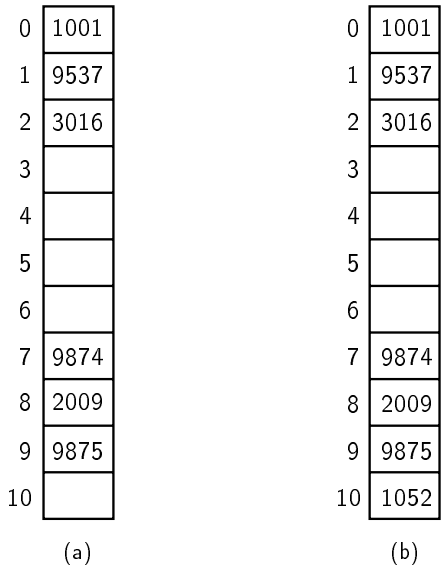
This is called **linear probing**.

Linear probing simply goes to the next slot in the table.

If the bottom is reached, wrap around to the top.

To avoid an infinite loop, one slot in the table must always be empty.

Linear Probing Example



Primary Clustering: Records tend to cluster in the table under linear probing since the probabilities for which slot to use next are not the same.

Improved Linear Probing

Instead of going to the next slot, skip by some constant c.

Warning: Pick M and c carefully.

The probe sequence SHOULD cycle through all slots of the table.

Pick c to be relatively prime to M.

There is still some clustering.

- Example: $c = 2$. $\mathbf{h}(k_1) = 3$. $\mathbf{h}(k_2) = 5$.
- The probe sequences for k_1 and k_2 are linked together.

Pseudo Random Probing

The ideal probe function would select the next slot on the probe sequence at random.

An actual probe function cannot operate randomly. (Why?)

Pseudo random probing:

- Select a (random) permutation of the numbers from 1 to $M-1$:

$$r_1, r_2, ..., r_{M-1}$$

- All insertions and searches use the same permutation.

Example: Hash table of size $M = 101$

- $r_1 = 2, r_2 = 5, r_3 = 32$.
- $\mathbf{h}(k_1) = 30, \mathbf{h}(k_2) = 28$.
- Probe sequence for k_1 is:
- Probe sequence for k_2 is:

Quadratic Probing

Set the i'th value in the probe sequence as

$$(\mathbf{h}(K) + i^2) \bmod M.$$

Example: M = 101.

- $\mathbf{h}(k_1) = 30$, $\mathbf{h}(k_2) = 29$.
- Probe sequence for k_1 is:
- Probe sequence for k_2 is:

Double Hashing

Pseudo random probing eliminates primary clustering.

If two keys hash to same slot, they follow the same probe sequence. This is called **secondary clustering**.

To avoid secondary clustering, need a probe sequence to be a function of the original key value, not just the home position.

Double hashing:

$$\mathbf{p}(K, i) = i * \mathbf{h}_2(K) \text{ for } 0 \leq i \leq M - 1.$$

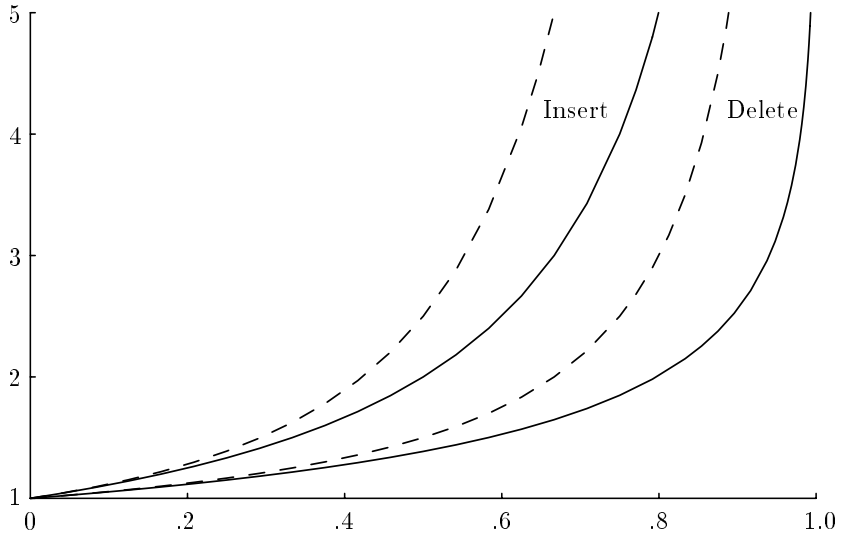
Be sure that all probe sequence constants are relatively prime to M.

Example: Hash table of size $M = 101$

- $\mathbf{h}(k_1) = 30, \mathbf{h}(k_2) = 28, \mathbf{h}(k_3) = 30$.
- $\mathbf{h}_2(k_1) = 2, \mathbf{h}_2(k_2) = 5, \mathbf{h}_2(k_3) = 5$.
- Probe sequence for k_1 is:
- Probe sequence for k_2 is:
- Probe sequence for k_3 is:

Analysis of Closed Hashing

The **load factor** is $\alpha = N/M$ where N is the number of records currently in the table.



Deletion

1. Deleting a record must not hinder later searches.
2. We do not want to make positions in the hash table unusable because of deletion.

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**.

A tombstone will not stop a search, but that slot can be used for future insertions.

Unfortunately, tombstones do add to the average path length.

Solutions:

1. Local reorganizations to try to shorten the average path length.
2. Periodically rehash the table (by order of most frequently accessed record).

Indexing

Goals:

- Store large files.
- Support multiple search keys.
- Support efficient insert, delete and range queries.

Entry sequenced file: Order records by time of insertion.

Use sequential search.

Index file: Organized, stores pointers to actual records.

Primary key: A unique identifier for records. May be inconvenient for search.

Secondary key: an alternate search key, often not unique for each record. Often used for search key.

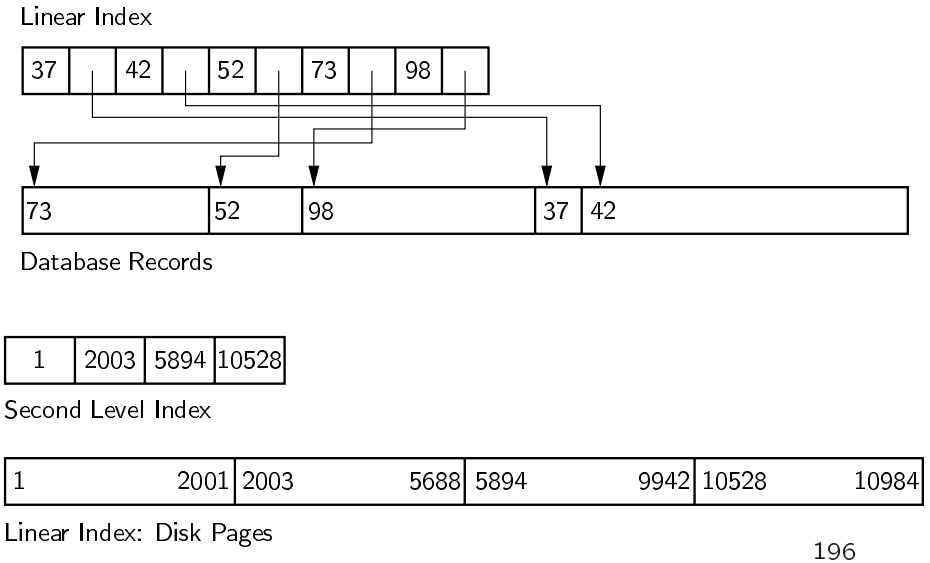
Linear Indexing

Linear Index: an index file organized as a simple sequence of key/record pointer pairs where the key values are in sorted order.

If the index is too large to fit in main memory, a second level index may be used.

Linear indexing is good for searching variable length records.

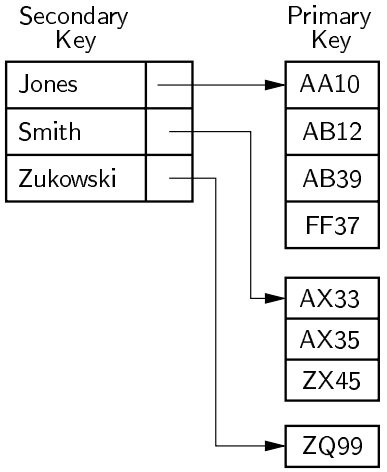
Linear indexing is poor for insert/delete.



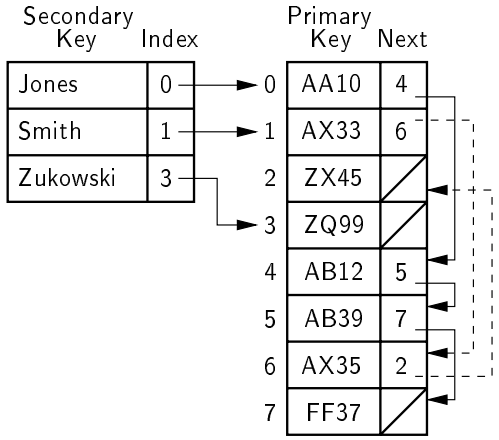
Inverted List

Inverted list is another term for a secondary index. A secondary key is associated with a primary key, which in turn locates the record.

Jones	AA10	AB12	AB39	FF37
Smith	AX33	AX35	ZX45	
Zukowski	ZQ99			



Inverted List (Continued)



Tree Indexing

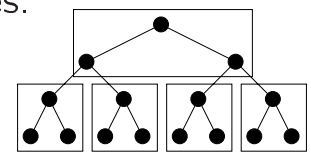
Linear index is poor for insertion/deletion.

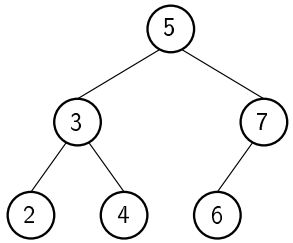
Tree index can efficiently support all desired operations:

- Insert/delete
- Multiple search keys
- Key range search

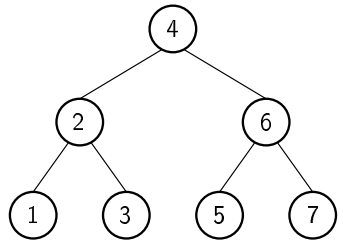
Storing a tree index on disk causes additional problems:

1. Tree must be balanced.
2. Each path from root to a leaf should cover few disk pages.





(a)



(b)

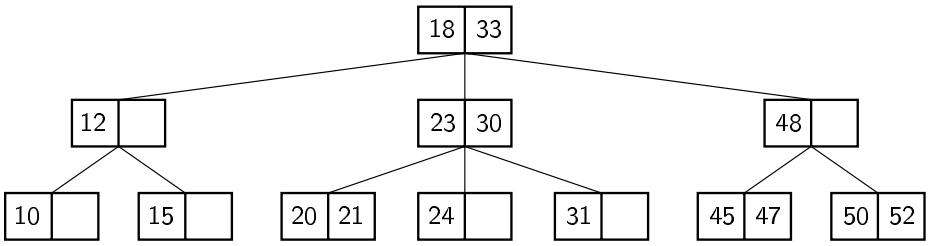
2-3 Tree

A 2-3 Tree has the following properties:

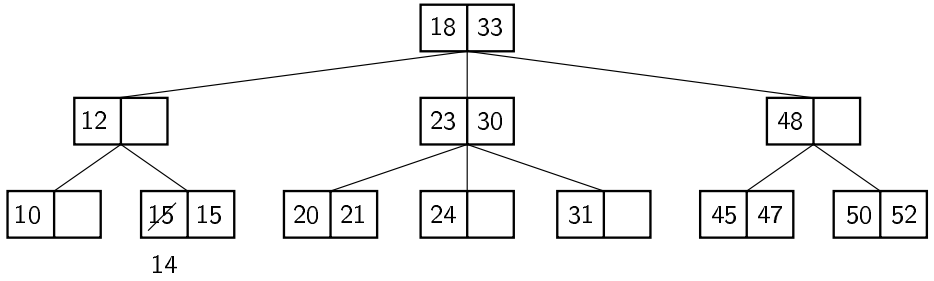
1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
3. All leaves are at the same level in the tree, so the tree is always height balanced.

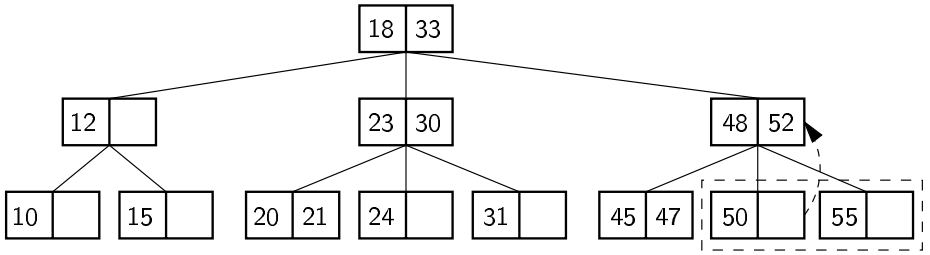
The 2-3 Tree also has a search tree property analogous to the BST.

The advantage of the 2-3 Treeover the BST is that it can be updated at low cost.

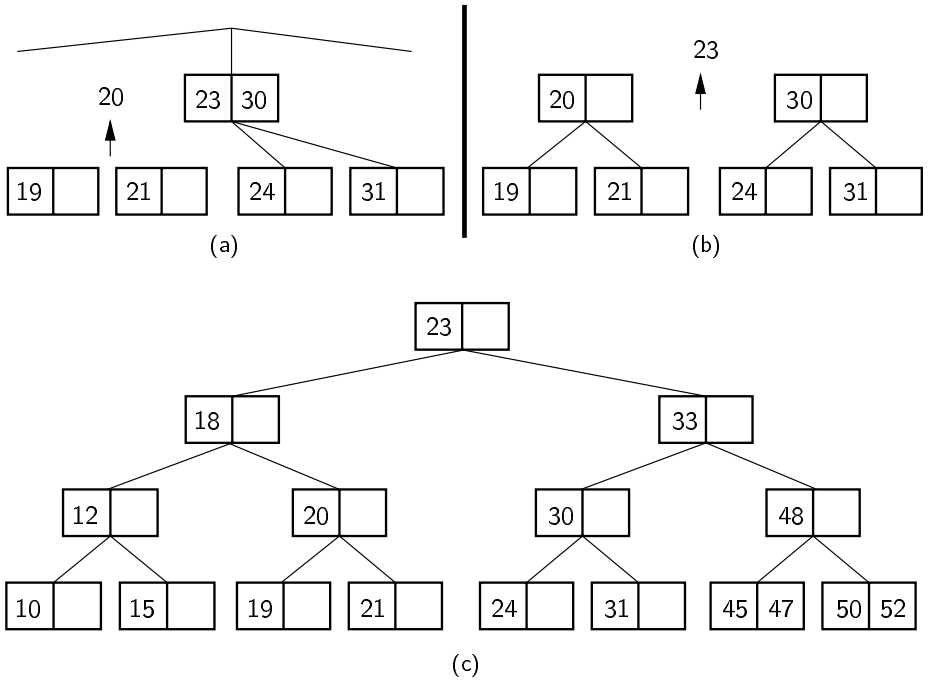


2-3 Tree Insertion





2-3 Tree Splitting



B-Trees

The B-Tree is an extension of the 2-3 Tree.

The B-Tree is now **the** standard file organization for applications requiring insertion, deletion and key range searches.

1. B-Trees are always balanced.
2. B-Trees keep related records on a disk page, which takes advantage of locality of reference.
3. B-Trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

B-Trees (Continued)

A B-Tree of order m has the following properties.

- The root is either a leaf or has at least two children.
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

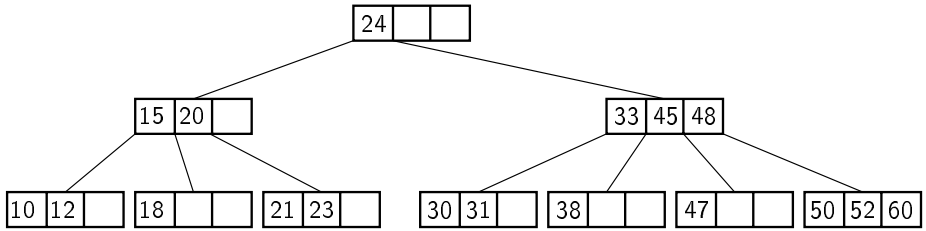
A B-Tree node is usually selected to match the size of a disk block.

A B-Tree node could have hundreds of children.

B-Tree Example

Search in a B-Tree is a generalization of search in a 2-3 Tree.

1. Perform a binary search on the keys in the current node. If the search key is found, then return the record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.



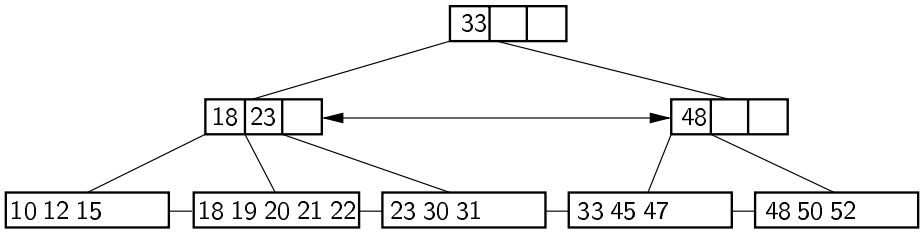
B^+-Trees

The most commonly implemented form of the B-Tree is the B^+-Tree.

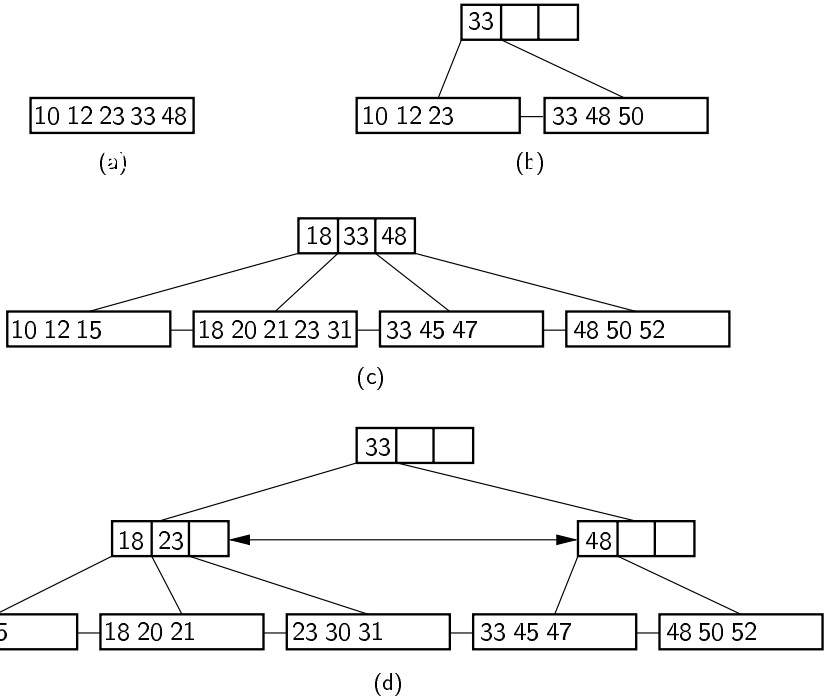
Internal nodes of the B^+-Tree do not store records – only key values to guide the search.

Leaf nodes store records or pointers to records.

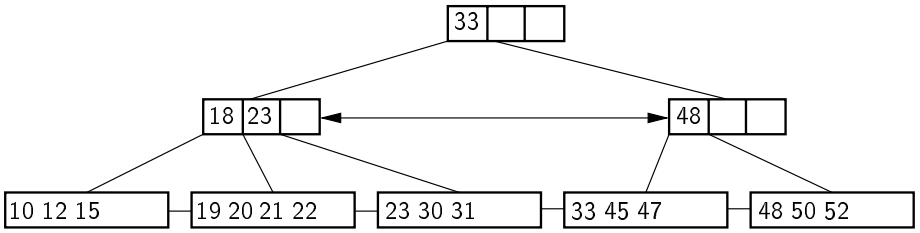
A leaf node may store more or less records than an internal node stores keys.

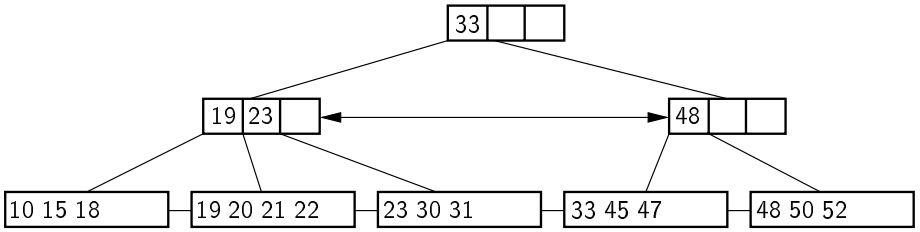


B^+-Tree Insertion



B^+-Tree Deletion





B-Tree Space Analysis

B^+-Tree nodes are always at least half full.

The B*-Tree splits two pages for three, and combines three pages into two. In this way, nodes are always 2/3 full.

Asymptotic cost of search, insertion and deletion of records from B-Trees, B^+-Trees and B*-Trees is $\Theta(\log n)$. (The base of the log is the (average) branching factor of the tree.)

Example: Consider a B^+-Tree of order 100 with leaf nodes containing 100 records.

1 level B^+-Tree:

2 level B^+-Tree:

3 level B^+-Tree:

4 level B^+-Tree:

Ways to reduce the number of disk fetches:

- Keep the upper levels in memory.
- Manage B^+-Tree pages with a buffer pool.