JICSCI803
Algorithms and Data Structures
September to December 2020

Highlights of Lecture 03

Proof Technique
Data Structures:
Introduction of Sorting algorithms

Algorithm Analysis Framework

Measuring an input's size
Measuring running time
Orders of growth (of the algorithm's efficiency function)
Worst-base, best-case and average-case efficiency

Algorithm and Data Structure

Relations
How to select data structure for an algorithm?

Worst Case Efficiency:

Efficiency (# of times the basic operation will be executed) for the worst case input of size n.

The algorithm runs the longest among all possible inputs of size n.

Best Case Efficiency:

Efficiency (# of times the basic operation will be executed) for the best case input of size n.

The algorithm runs the fastest among all possible inputs of size n.

Average Case Efficiency:

Efficiency (#of times the basic operation will be executed) for a typical/random input of size n.

NOT the average of worst and best case

How to find the average case efficiency?

Proof Techniques

- Proof by contradiction (reduction to absurdity, *reductio ad absurdum* in Latin)
- Proof by induction
- Proof (of falsity) by counterexample

Proof Techniques

- Proof by contradiction (*reductio ad absurdum*)
  - Assume the proposition is false
  - Show that this leads to a contradiction
  - Therefore the proposition must be true

Proof Techniques

- Proof by contradiction, example:
  - **Proposition**: *There are an infinite number of prime numbers*
  - **Proof:**
    1. Assume that there are a finite number of primes $P_1, P_2 \ldots, P_n$
    2. Let $Q = P_1 \times P_2 \times \ldots \times P_n + 1$
    3. Note that Q is not evenly divisible by any of $P_1, P_2 \ldots, P_n$
    4. Thus either Q is prime or it is divisible by some prime not in $P_1, P_2 \ldots, P_n$
    5. This contradicts our assumption in 1. that we have listed all the primes
    6. Therefore there are an infinite number of primes

Proof Techniques

- Proof by generalized induction

- Demonstrate that $P(n)$ holds for all $a \leq n < b$
- For any integer $k \geq b$ show that $P(k)$ can be proven from the assumption that $P(m)$ is true for all $a \leq m < b$

Proof Techniques

- Proof by generalized induction, example:

Proposition: Every positive composite integer can be expressed as a product of prime numbers

Proof:

1. Consider the smallest case $n = 4$, the proposition is clearly true in this case ($4 = 2 \times 2$)
2. Consider a case where $n > 4$
3. Assume the proposition is true for all m such that $4 \leq m < n$
4. Let $d > 1$ be the smallest divisor of n; d must be prime (proof ?)
5. Let $q = n / d$
6. If q is prime we have $n = q \times d$ where both q and d are prime, if q is composite then q is expressible as a product of prime numbers
7. In either case, n is expressible as a product of prime numbers

QED

Proof Techniques

- Proof (of falsity) by counterexample

- Find a value k for which $P(k)$ is false
- Clearly the assertion that $P(n)$ is true for all n must be false because it is not true for $P(k)$

Proof Ttechniques

- Proof by counter example, example:
  - We note the following relation:

n	n^3+2	2^n
1	3	2
2	10	4
3	29	8
4	66	16

Proof Techniques

- Proof by counterexample, example:
  1. We propose that $n^3+2 > 2^n$ for all n, $P(n)$
  2. We note that 10^3+2 (1002) $< 2^{10}$(1024)
  3. Therefore $\sim P(n)$, ($P(n)$ is false)
  - In fact, $n^3+2 < 2^n$ for all $n > 9$ (proof ?)

EXPONENTS

Exponents

- **Definition for A > 0:**
  $X^A = X * X * X \ldots. * X$

  ← X appears A times

- $X^A \times X^B = X^{A+B}$
- $X^A / X^B = X^{A-B}$
- $(X^A)^B = X^{AB}$
- $X^N + X^N = 2X^N \neq X^{2N}$
- $2^N + 2^N = 2^{N+1}$

Logarithms

- Logarithms
  - **Definition:** $X^A = B$ **iff** $\log_X B = A$

  $\log_X B =$ *"the power you raise X to get B"*

Logarithms

- Logarithms
  - **Definition:** $X^A = B$ **iff** $\log_X B = A$

  - **Theorem:**

$\log_A B = \log_C B / \log_C A$; $A, B, C > 0$, $A \neq 1$

  - **Proof :**

Let $X = \log_C B$, $Y = \log_C A$ and $Z = \log_A B$.

  - From the definition, $C^X = B$, $C^Y = A$ and $A^Z = B$. Combining these gives $B = C^X = A^Z = (C^Y)^Z = C^{YZ}$. Therefore $X = YZ$ which leads to $Z = X / Y$

Summation Formulas

Let N $\geq$ 0, let A, B, and C be constants, and let *f* and *g* be any functions. Then:

$$\boxed{\sum_{k=1}^{N} Cf(k) = C\sum_{k=1}^{N} f(k)}$$

S1: factor out constant

$$\boxed{\sum_{k=1}^{N} (f(k) \pm g(k)) = \sum_{k=1}^{N} f(k) \pm \sum_{k=1}^{N} g(k)}$$

S2: separate summed terms

$$\boxed{\sum_{k=1}^{N} C = NC}$$

S3: sum of constant

$$\boxed{\sum_{k=1}^{N} k = \frac{N(N+1)}{2}}$$

S4: sum of *k*

$$\boxed{\sum_{k=1}^{N} k^2 = \frac{N(N+1)(2N+1)}{6}}$$

S5: sum of *k* squared

$$\boxed{\sum_{k=0}^{N} 2^k = 2^{N+1} - 1}$$

S6: sum of 2^*k*

$$\boxed{\sum_{k=1}^{N} k2^{k-1} = (N-1)2^N + 1}$$

S7: sum of k2^(*k-1*)

Data Structure

In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently.

- A data structure is composed of data representation and its associated operation.

(1) Data representation

More typically, a data structure is meant to be an organization or structuring for a collection of data items.

(2) Associated operation

Such as : search, print, modify, sort, etc.

Abstract Data Type, an example

```
ADT
{
    data object;
    relations of data element;
    operations;
};
```

Abstract Data Type, an example

ADT Complex_Number {

Objects: $\{a_1+a_2i \mid a_1, a_2i \in R\}$

where R is the set of real

Operations: let $x= x_1+x_2i$, $y= y_1+y_2i$

real realpart(x): get the real part of x

real imagpart(x): get the imaginary part of x

complex_number add(x,y): return x+y,

that is $(x_1+y_1)+(x_2+y_2)i$

complex_number subtract(x,y): return x-y,

that is $(x_1-y_1)+(x_2-y_2)i$

complex_number multiplay(x,y): return x*y,

that is $(x_1 \cdot y_1-x_2{}^*y_2)+(x_1{}^*y_2+x_2{}^*y_1)i$

}

Data Structure - costs and benefits:

Each data structure has associated cost and benefits.

Cost: A data structure requires a certain amount of space to store each data item and a certain amount of time to perform single basic operation, and a certain amount of programming effort.

Benefit: Using the right data structure can solve a problem efficiently, say, meeting the requirements of limited resources, for example time.

Data Structure—3 Steps for Selection

- Analyze problem, determine the resource constraints.

- Determine the basic operations that must be supported and quantify the resource constraints for each operation.

- Select the data structure that best meets these requirements.

Some Common Data Structures (DSs)

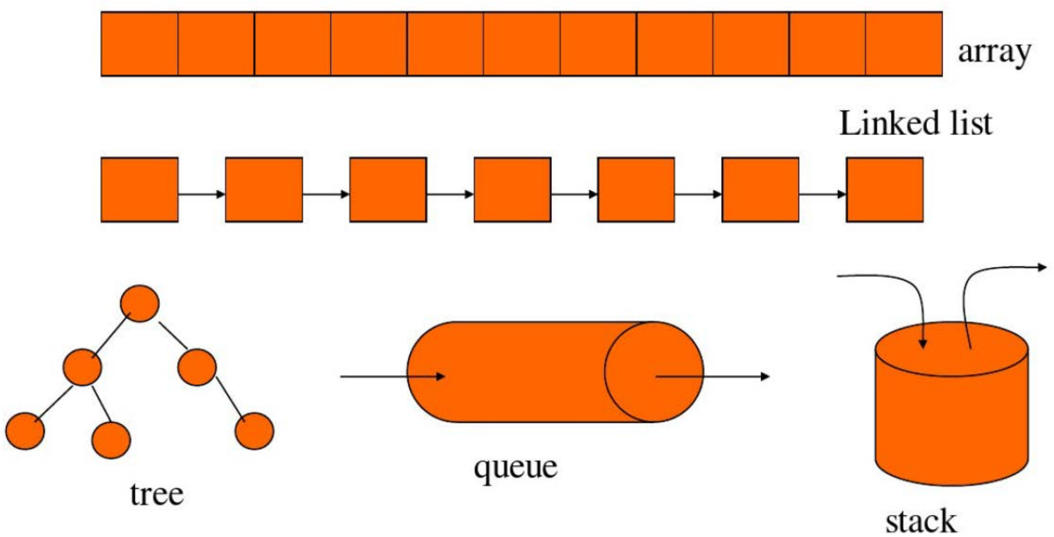
-– Linear DS: Arrays and Lists (Stacks and Queues

—Records (structures)

-–Graphs (various graphs)

-–Trees

-– Associative Tables (hash tables / dictionaries)

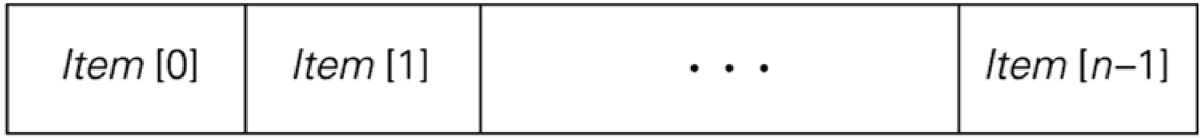


Arrays

An array is a data structure consisting of a fixed number of data items of the same type

E.g. table: array[1..50] of integer,

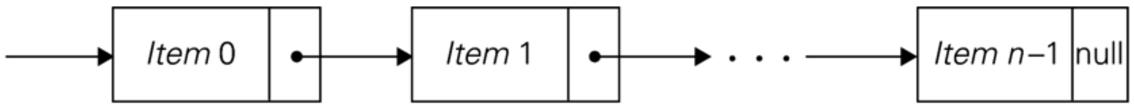
letters: array[1..26] of character



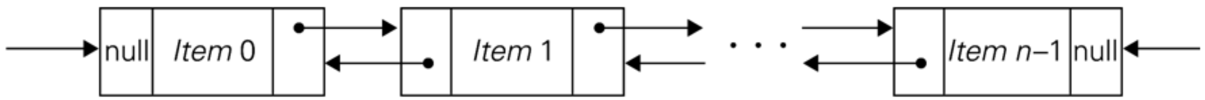
Array of *n* elements

Linked lists

- A sequence of zero or more nodes each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list.
- Singly linked list (next pointer)
- Doubly linked list (next + previous pointers)
- Two special types of lists: Stack and queue.



Singly linked list of n elements



Double linked list of n elements

Stacks

– A stack is a data structure which holds multiple elements of a single type

– Elements can be removed from a stack only in the reverse order to that in which they were inserted (LIFO =: Last In First Out)

– A stack can be implemented with an array and an integer counter to indicate the current number of elements in the stack

Stacks

– E.g. stack: array[1..50] of integer
ctr: integer
ctr = 0

– To put an element on the stack
procedure push(elt)
stack[ctr] = elt
ctr = ctr + 1

– To remove an element from the stack
if ctr = 0 then elt = nil
else elt=stack[ctr] and ctr = ctr – 1

Queues

– A queue is a data structure which holds multiple elements of a single type

– Elements can be removed from a queue only in the order in which they were inserted (FIFO =: First In First Out)

– A queue can be implemented with an array and two integer counter to indicate the current start and next insertion positions

Queues

– A queue is a data structure which holds multiple elements of a single type

– Elements can be removed from a queue only in the order in which they were inserted (FIFO =: First In First Out)

– A queue can be implemented with an array and two integer counter to indicate the current start and next insertion positions

Queues

– E.g. queue: array[1..50] of integer

```
start: integer
next: integer
start = 1; next = 1
```

– To put an element in the queue

```
procedure enqueue(elt)
queue[next] = elt
next = next+1
if next > 50 then next = 1
```

– To take an element out of the queue

```
procedure dequeue(elt)
    if start = next then elt = nil
     Else
        elt = queue[start]
        start = start +1
        if start > 50 then start = 1
```

- Records (Structures)
  - A record is a data structure consisting of a fixed number of items
  - Unlike an array, the elements in a record may be of differing types and are named.
  - E.g. type person = record

```
                name: string
                age: integer
                height: real
                female: Boolean
                children: array[1:10] of string
```

- Records (Structures)
  - A record is a data structure consisting of a fixed number of items
  - Unlike an array, the elements in a record may be of differing types and are named.
  - E.g. type person = record

```
                 name: string
                 age: integer
                 height: real
                 female: Boolean
                 children: array[1:10] of string
```

- Records (Structures)
  - An array may appear as a field in a record
  - Records may appear as elements of an array
  - E.g. staff: array[1..50] of person
  - Records are typically addressed by a pointer
  - E.g. type boss = ^person declares boss to be a pointer to records of type person
  - Fields of a record are accessible via the field name
  - E.g. staff[5].age, boss^.name

Graphs

Formal definition

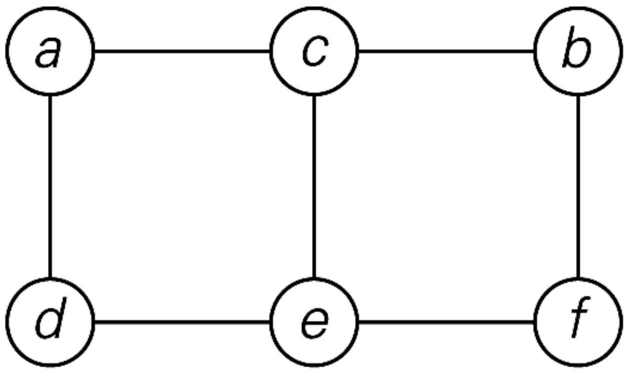
A graph *G = <V, E>* is defined by a pair of two sets: a finite set V of items called **vertices** and a set E of vertex pairs called **edges**.

Undirected and **directed** graphs (**digraph**).

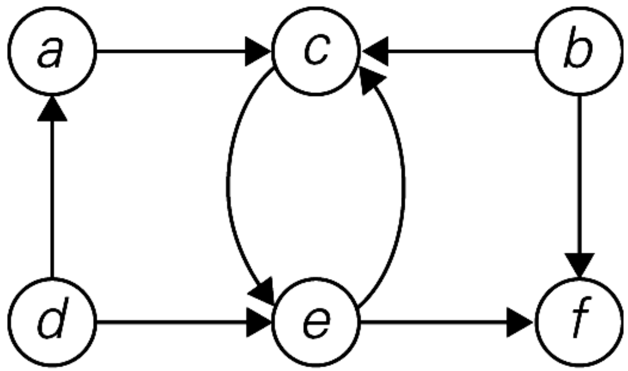
What's the maximum number of edges in an undirected graph with |V| vertices?

Complete, **dense**, and **sparse** graph

A graph with every pair of its vertices connected by an edge is called complete. $K_{|V|}$



(a)



(b)

Graph Representation

Adjacency matrix

n x n boolean matrix if |V| is n.

The element on the ith row and jth column is 1 if there's an edge from ith vertex to the jth vertex; otherwise 0.

The adjacency matrix of an undirected graph is symmetric.

$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array}
\begin{array}{c}
\begin{array}{cccccc} a & b & c & d & e & f \end{array} \\
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

Adjacency linked lists

A collection of linked lists, one for each vertex, that contain all the vertices adjacent to the list's vertex.

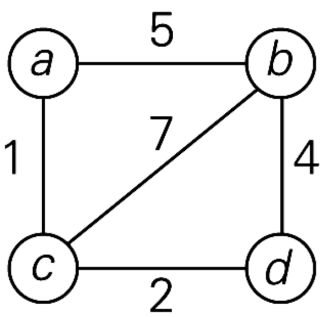
$$
\begin{array}{|c|l}
\hline
a & \to c \to d \\
\hline
b & \to c \to f \\
\hline
c & \to a \to b \to e \\
\hline
d & \to a \to e \\
\hline
e & \to c \to d \to f \\
\hline
f & \to b \to e \\
\hline
\end{array}
$$

Which data structure would you use if the graph is a 100-node star shape?

Graphs

Weighted graphs

Graphs or digraphs with numbers assigned to the edges.



(a)

$$
\begin{array}{c} \\ a \\ b \\ c \\ d \end{array}
\begin{array}{c}
\begin{array}{cccc} a & b & c & d \end{array} \\
\begin{bmatrix} \infty & 5 & 1 & \infty \\ 5 & \infty & 7 & 4 \\ 1 & 7 & \infty & 2 \\ \infty & 4 & 2 & \infty \end{bmatrix}
\end{array}
$$

(b)

a	$\rightarrow b, 5 \rightarrow c, 1$
b	$\rightarrow a, 5 \rightarrow c, 7 \rightarrow d, 4$
c	$\rightarrow a, 1 \rightarrow b, 7 \rightarrow d, 2$
d	$\rightarrow b, 4 \rightarrow c, 2$

(c)

(a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Graph Properties -- Paths and Connectivity

Paths

A path from vertex u to v of a graph G is defined as a sequence of adjacent (connected by an edge) vertices that starts with u and ends with v.

Simple paths: All edges of a path are distinct.

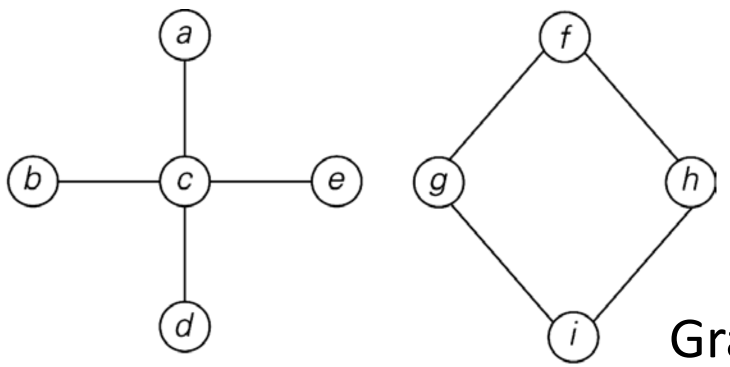
Path lengths: the number of edges, or the number of vertices – 1.

Connected graphs

A graph is said to be connected if for every pair of its vertices u and v there is a path from u to v.

Connected component

The maximum connected subgraph of a given graph.



Graph that is not connected

Graph Properties -- Acyclicity

Cycle

A simple path of a positive length that starts and ends a the same vertex.

Acyclic graph

A graph without cycles

DAG (Directed Acyclic Graph)

Trees

Trees

A tree (or free tree) is a connected acyclic graph.

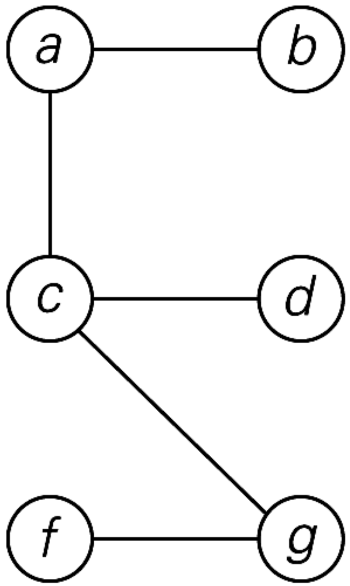
Forests: a graph that has no cycles but is not necessarily connected.

Properties of trees

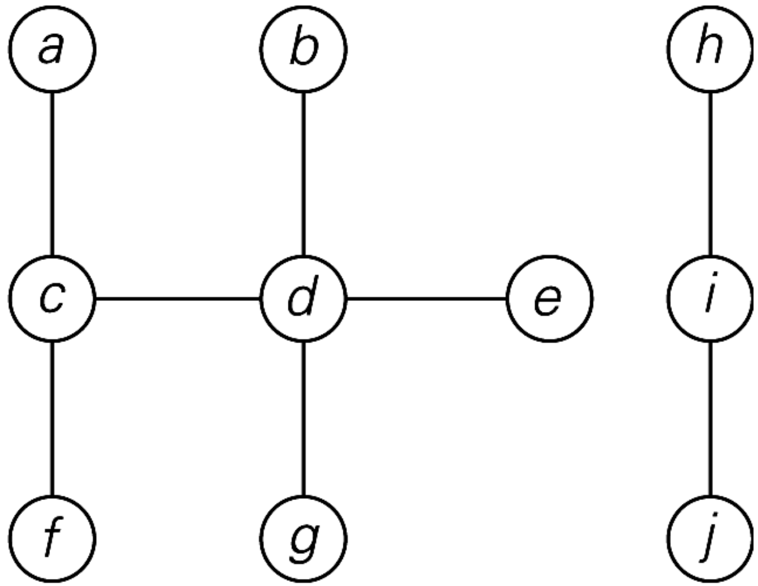
- |E| = |V| - 1
- For every two vertices in a tree there always exists exactly one simple path from one of these vertices to the other. Why?

  Rooted trees: The above property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree.

  Levels of rooted tree.



(a)



(b)

A tree and a forest

Rooted Trees

ancestors

For any vertex *v* in a tree *T*, all the vertices on the simple path from the root to that vertex are called ancestors.

descendants

All the vertices for which a vertex *v* is an ancestor are said to be descendants of *v*.

parent, child and siblings

If (*u, v*) is the last edge of the simple path from the root to vertex *v* (and $u \neq v$), *u* is said to be the parent of *v* and *v* is called a child of *u*.

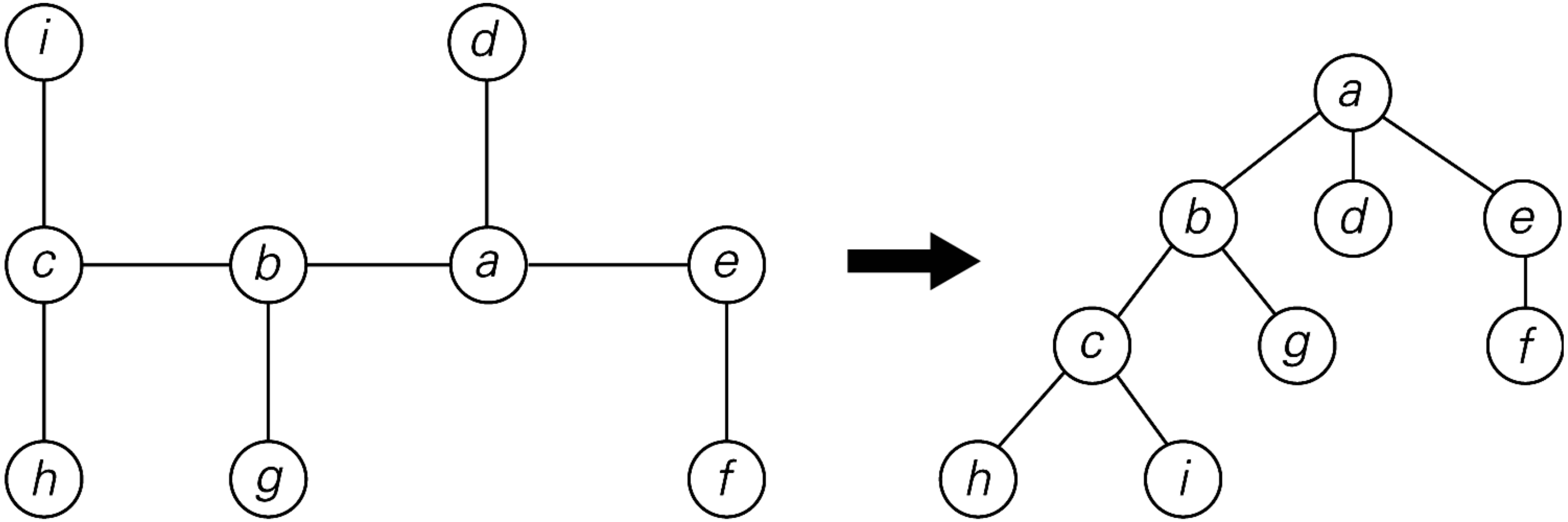
Vertices that have the same parent are called siblings.

Leaves

A vertex without children is called a leaf.

Subtree

A vertex *v* with all its descendants is called the subtree of *T* rooted at *v*.



(a) (b)

Transformation of a free tree into a rooted tree

Tree Depth and Height

Depth of a vertex

The length of the simple path from the root to the vertex.

Height of a tree

The length of the longest simple path from the root to a leaf.

Ordered Trees

Ordered trees

An ordered tree is a rooted tree in which all the children of each vertex are ordered.

Binary trees

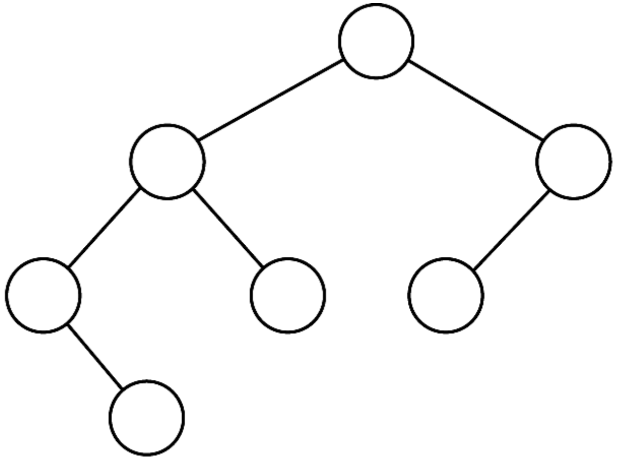
A binary tree is an ordered tree in which every vertex has no more than two children and each children is designated as either a left child or a right child of its parent.

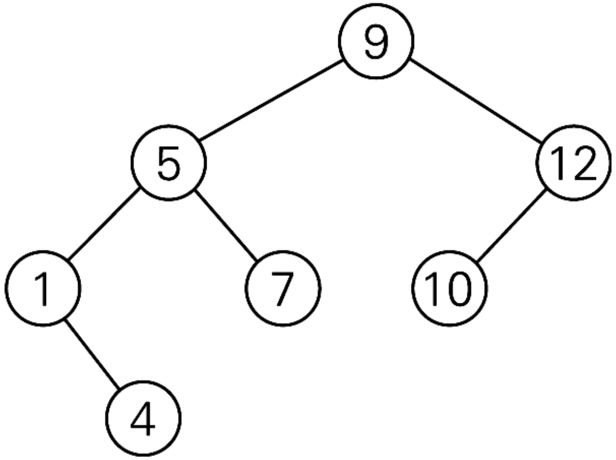
Binary search trees=:

Each vertex is assigned a number.

A number assigned to each parental vertex is larger than all the numbers in its left subtree and smaller than all the numbers in its right subtree.

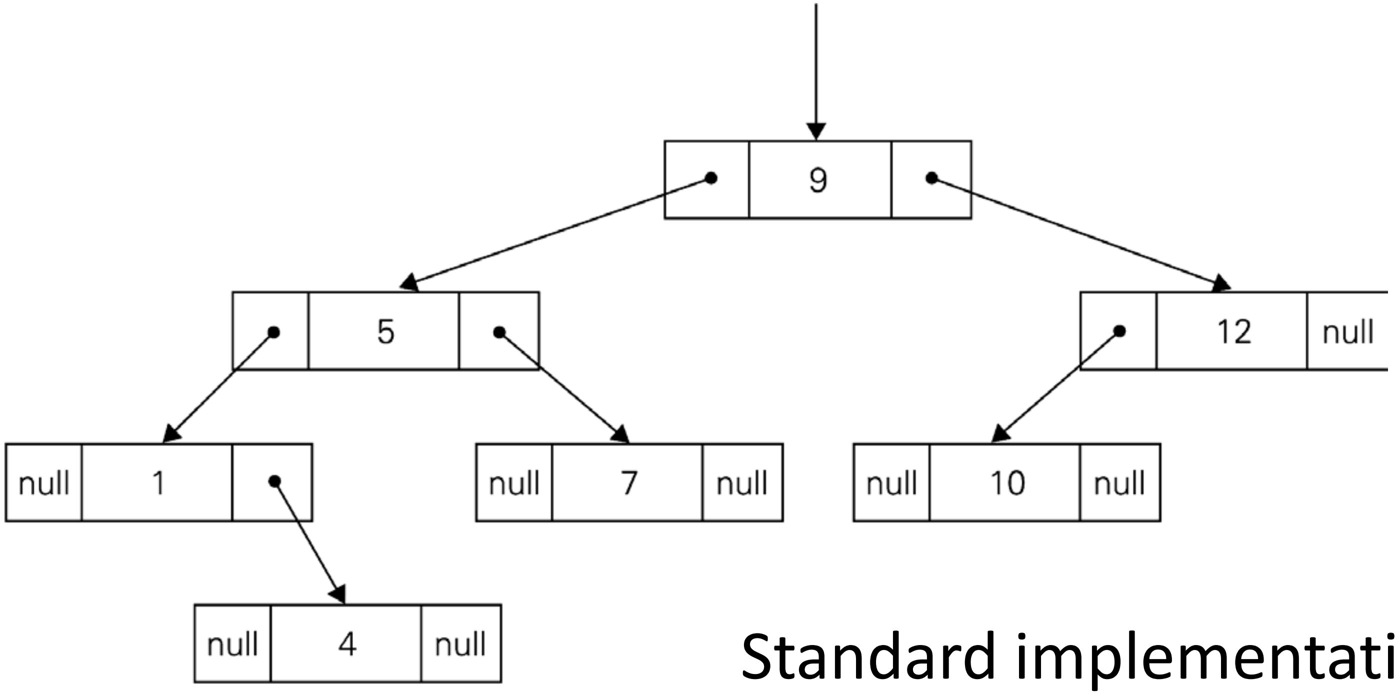
$\lfloor \log_2 n \rfloor \leq h \leq n - 1$, where h is the height of a binary tree.



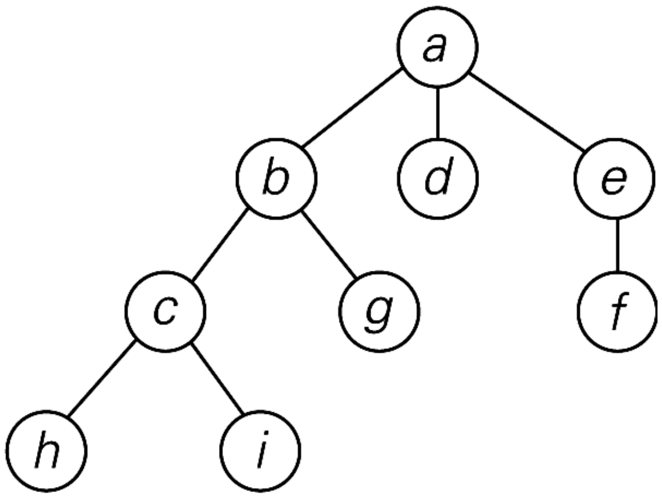


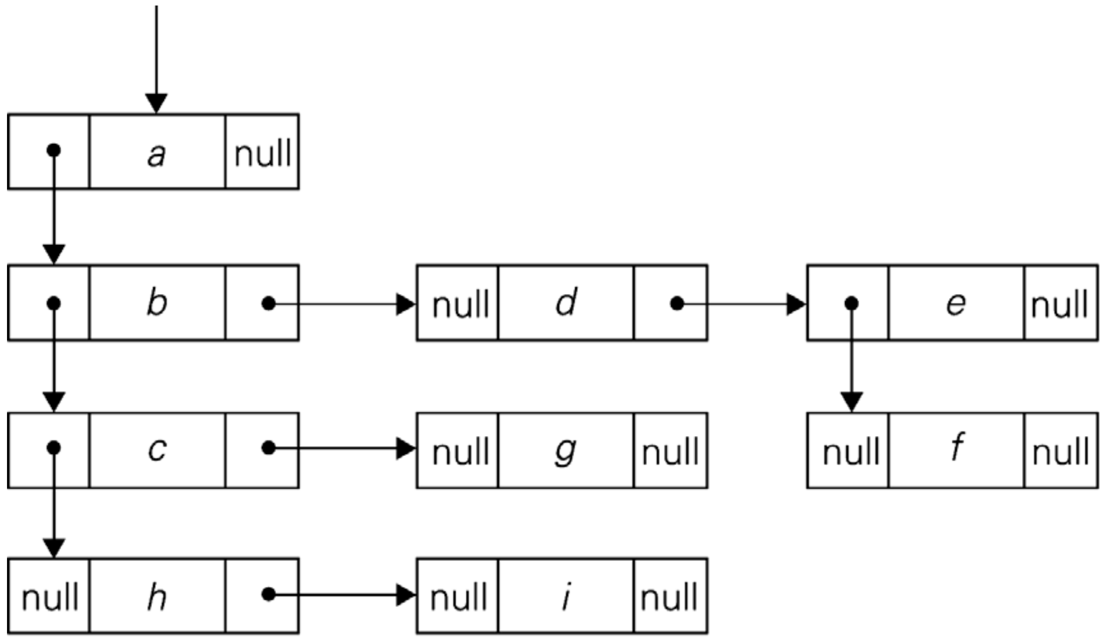
(a) (b)

A binary tree and a binary search tree



Standard implementation of (b)





A rooted tree and its first child-next sibling representation

- Heaps
  - A heap is a binary tree with an additional property
  - The value in any node is less than or equal to the value in its parent node. (except for the root node).
  - We can store a heap (or any other binary tree) in an array

- Heap in an array
  - Heap[1] is the root of the tree
  - Heap[2] and Heap[3] are the children of Heap[1]
  - In general, Heap[i] has children Heap[2i] and Heap[2i+1]

- Operations on heaps
  - If we have a non-heap how can we convert it into one?
  - If we have a heap and add a new element how can we restore the heap property?
  - If we have a heap and remove an element how can we restore the heap property?

- Operations on heaps
  - We need two basic functions to manage heaps:
    - siftup
    - siftdown
  - Each compares an element of the heap with other elements.

Sift-up
Input Array H[1...n] and the index between 1 and n.
Output Sift-up H[i]（if necessary）
<to make it is larger than its parent node>

Algorithm Description
done ← **false**
if i = 1 **then exit** {node i root}
repeat
    if key(H[i]) > key(H[⌊i/2⌋]) **then** swap H[i] and H[⌊i/2⌋]
    else done ← **true**
    i ← ⌊i/2⌋
until i = 1 **or** done

Sift-down
Input Array H[1...n] and the index between 1 and n.
Output Sift-down H[i]（if necessary）
<to make it is smaller than its descent node>

Algorithm Description
done ← **false**
if 2i > n **then exit** {node i is leave }
repeat
  i ← 2i
  if i + 1 ≤ n **and** key(H[i+1]) > key(H[i]) then i ← i + 1
  if key(H[⌊i/2⌋]) < key(H[i]) **then** swap H[i] and H[⌊i/2⌋]
  else done ← **true**
  end if
until 2i > n **or** done

- Associative tables
  - An associative table behaves like an array with no restriction on index value
  - Unlike an array, there is no guarantee that item access is $\Theta(1)$
  - Storage is not necessarily required for unreferenced items

- Associative tables
  - Implementation via a list:

    type table_list = ^table_node

    type table_node = record

        index: index_type

        value: stuff

        next: ^table_node

  - Access to A[“fred”] is accomplished by marching through the list until fred is found in the index field or end of list is reached
  - Access is $\Theta(n)$

- Associative tables
  - Implementation via an array:

    type table_array = array[1..size] of ^table_node

    type table_node = record

      index: index_type

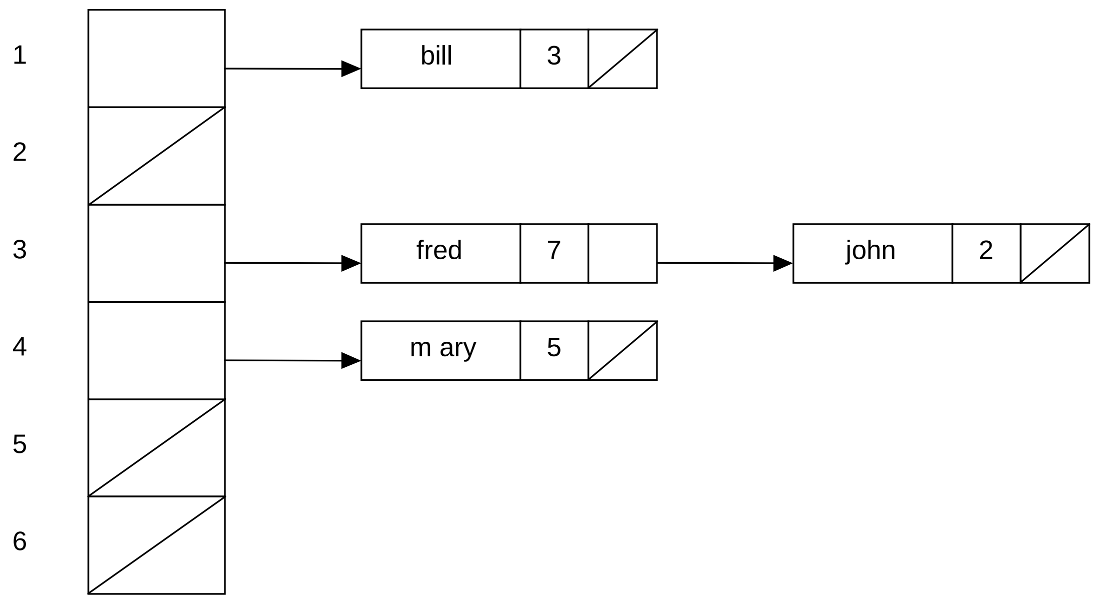
      value: stuff

      next: ^table_node

  - Access to A[“fred”] is via the array and is accomplished by constructing an index h based on “fred” which is then used to access a (list of) node(s) via the table_array

- Associative tables
  - – Implementation via an array:
  - – E.g. A[“fred”]
    - h(“fred”) = 3
    - ptr = table_array[3]
    - step through list starting at ptr looking for “fred”
  - – Note that h(“fred”) need not be a unique value
  - – However h( ) should generate a good even spread of values in 1..size

- Associative tables
  - E.g. size = 6,
    h(“bill”) = 1, h(“fred”) = 3, h(“mary”) = 4, h(“john”) = 3

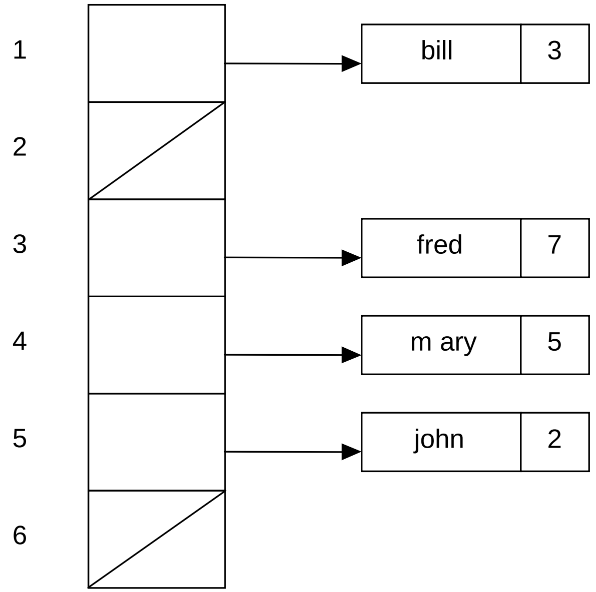


- Associative tables
  - Implementation via an array, an alternative approach:
    - On insertion into the array of a new item pointer, if the array entry is already occupied we insert into the next available space in the array.
    - Searching is now accomplished by comparing the key values from h(key) onwards in the array until a match or an empty slot is found
    - The efficiency of this scheme depends on the fullness of the array as well as on the evenness of the hash function

- Associative tables
  - Implementation via an array, an alternative approach:

```
type table_array = array[1..size] of ^table_node
type table_node = record
    index: index_type
    value: stuff
```

- Associative tables
  - E.g. size = 6,
    h("bill") = 1, h("fred") = 3, h("mary") = 4, h("john") = 3



- Associative tables
  - Insertion into an associative table is efficient;
  - Finding an entry in an associative table is efficient;
  - Deletion from an associative table is efficient; although some care must be taken in handling deletions from the second array based implementation
  - Listing of entries (especially ordered listing) is not efficient

Sorting - Introduction

- Rearrange the items of a given list in (ascending/descending) order.

  Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$

  Output: A reordering $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

- Why sorting?

  Help searching

  Algorithms often use sorting as a key subroutine.

- Sorting key

  A specially chosen piece of information used to guide sorting. I.e., sort student records by names.

Sorting - Introduction

How to sort n numbers? $<a_1, \ a_2, \ldots, a_n>$

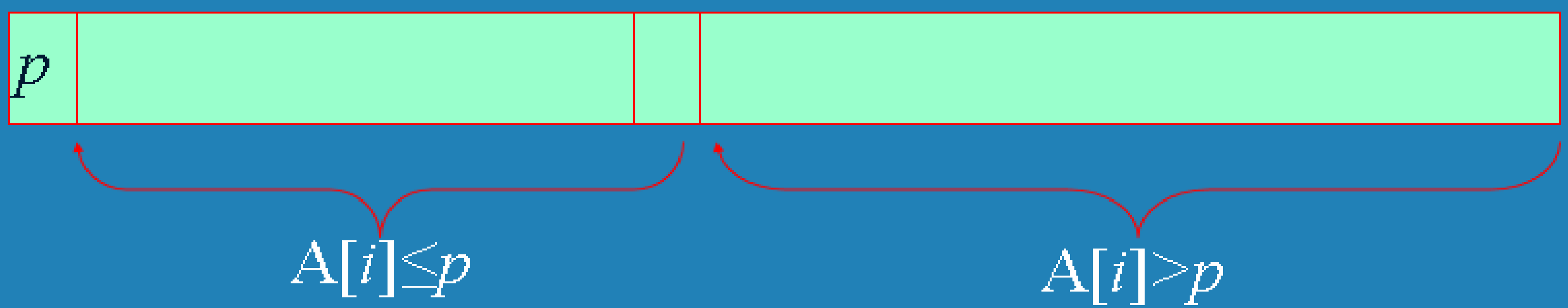
Sorting: Introduction

- Sorting algorithms are Divide and Conquer examples.
- D&C is the most well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instance sets
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Quick sort, an example

- Select a *pivot* (partitioning element)
- Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than the pivot (See algorithm *Partition* in section 4.2)
- Exchange the pivot with the last element in the first (i.e., ≤ sublist) – the pivot is now in its final position
- Sort the two sublists



Quick sort, the efficiency

Best case: split in the middle — $\Theta(\ n \log n)$
Worst case: sorted array! — $\Theta(\ n^2)$
Average case: random arrays — $\Theta(\ n \log n)$

- Some new sorts:
  - As you saw quick sort is normally in O(n log n) but in the worst case it is still in $O(n^2)$.
  - There are some sort algorithms that are in O(n log n) even for the worst case behaviour.
  - Let us look at a couple of these.

Heapsort

– Heapsort sorts an array by using the heap property we saw earlier.

- Heapsort

```
Procedure heapsort(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
       swap T[1] and T[i]
        siftdown(T[1 .. i – 1], 1)
```

- Heapsort

```
procedure makeheap(T[1..n])
   for i = n ÷ 2 to 1 step –1 do
          siftdown(T, i)
    end for
end
```

- Heapsort

T = [7, 2, 9, 5, 1, 3, 8, 4] makeheap

[7, 2, 9, *5*, 1, 3, 8, *4*] siftdown 5 needs 0 swaps

[7, 2, *9*, 5, 1, *3*, *8*, 4] siftdown 9 needs 0 swaps

[7, *2*, *9*, *5*, 1, 3, 8, 4] siftdown 2 needs 2 swaps

[*7*, *5*, *9*, 4, 1, 3, 8, 2] siftdown 7 needs 2 swaps

[9, 5, 8, 4, 1, 3, 7, 2] heap complete

- Heapsort

T = [7, 2, 9, 5, 1, 3, 8, 4]

[9, 5, 8, 4, 1, 3, 7, 2] after makeheap

[2, 5, 8, 4, 1, 3, 7, **9**] swap 2 and 9

[8, 5, 7, 4, 1, 3, 2, **9**] siftdown 2

[2, 5, 7, 4, 1, 3, **8**, **9**] swap 2 and 8

[7, 5, 3, 4, 1, 2, **8**, **9**] siftdown 2

[2, 5, 3, 4, 1, **7**, **8**, **9**] swap 2 and 7

[5, 4, 3, 2, 1, **7**, **8**, **9**] siftdown 2

[1, 4, 3, 2, **5**, **7**, **8**, **9**] swap 1 and 5

[4, 2, 3, 1, **5**, **7**, **8**, **9**] siftdown 1

[1, 2, 3, **4**, **5**, **7**, **8**, **9**] swap 1 and 4

[3, 2, 1, **4**, **5**, **7**, **8**, **9**] siftdown 1

[1, 2, **3**, **4**, **5**, **7**, **8**, **9**] swap 1 and 3

[2, 1, **3**, **4**, **5**, **7**, **8**, **9**] siftdown 1

[1, **2**, **3**, **4**, **5**, **7**, **8**, **9**] swap 1 and 2 - sorted

- Heapsort

$\Theta(n \log n)$

  - Siftdown is in $\Theta(\log n)$
  - Heapsort is in $\Theta(n \log n) + \Theta((n - 1) \log n = \Theta(n \log n)$

– Makeheap is in

- Heapsort
  - Makeheap is in $\Theta(n \log n)$
  - Siftdown is in $\Theta(\log n)$
  - Heapsort is in $\Theta(n \log n) + \Theta((n - 1) \log n = \Theta(n \log n)$

- Heapsort
  - Makeheap is in $\Theta(n \log n)$
  - Siftdown is in $\Theta(\log n)$
  - Heapsort is in $\Theta(n \log n) + \Theta((n - 1) \log n = \Theta(n \log n)$

- Heapsort
  - Makeheap is in $\Theta(n \log n)$
  - Siftdown is in $\Theta(\log n)$
  - Heapsort is in $\Theta(n \log n) + \Theta((n - 1) \log n = \Theta(n \log n)$

- Mergesort
-

```
global X[1..n]    // temporary array used in merge procedure

procedure mergesort(T[left..right])
   if left < right then
      centre = (left + right) ÷ 2
      mergesort(T[left..centre])
      mergesort(T[centre+1..right])
      merge(T[left..centre], T[centre+1..right],T[left..right])
```

- Mergesort

```
procedure merge(A[1..a], B[1..b], C[1..a + b])
   apos = 1; bpos = 1; cpos = 1
   while apos ≤ a and bpos ≤ b do
      if A[apos] ≤ B[bpos] then
         X[cpos] = A[apos]
          apos = apos + 1; cpos = cpos + 1
      else
         X[cpos] = B[bpos]
         bpos = bpos + 1; cpos = cpos + 1
   while apos ≤ a do
      X[cpos] = A[apos]
      apos = apos + 1; cpos = cpos + 1
   while bpos ≤ b do
      X[cpos] = B[bpos]
      bpos = bpos + 1; cpos = cpos + 1
   for cpos = 1 to a + b do
      C[cpos] = X[cpos]
```

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
    [7, 2, 9, 5] mergesort T[1..2]
    [7, 2] mergesort T[1..1]
    [7] done
    [**7**, 2] mergesort T[2..2]
    [2] done

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done
[**2**, **7**, 9, 5] mergesort T[3..4]
[9, 5] mergesort T[3..3]
[9] done
[**9**, 5] mergesort T[4..4]
[5] done
[9] [5] merge
[5, 9] done
[2, 7] [5, 9] merge
[2, 5, 7, 9] done

T = [**2**, **5**, **7**, **9**, 1, 3, 8, 4] mergesort T[5..8]
[1, 3, 8, 4] mergesort T[5..6]
[1, 3] mergesort T[5..5]
[1] done
[**1**, 3] mergesort T[6..6]
[3] done
[1] [3] merge
[1, 3] done
[**1**, **3**, 8, 4] mergesort T[7..8]
[8, 4] mergesort T[7..7]
[8] done
[**8**, 4] mergesort T[8..8]
[4] done

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done
[**2**, **7**, 9, 5] mergesort T[3..4]
[9, 5] mergesort T[3..3]
[9] done
[**9**, 5] mergesort T[4..4]
[5] done
[9] [5] merge
[5, 9] done
[2, 7] [5, 9] merge
[2, 5, 7, 9] done

T = [**2**, **5**, **7**, **9**, 1, 3, 8, 4] mergesort T[5..8]
[1, 3, 8, 4] mergesort T[5..6]
[1, 3] mergesort T[5..5]
[1] done
[**1**, 3] mergesort T[6..6]
[3] done
[1] [3] merge
[1, 3] done
[**1**, **3**, 8, 4] mergesort T[7..8]
[8, 4] mergesort T[7..7]
[8] done
[**8**, 4] mergesort T[8..8]
[4] done
[8] [4] merge
[4, 8] done

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done
[**2**, **7**, 9, 5] mergesort T[3..4]
[9, 5] mergesort T[3..3]
[9] done
[**9**, 5] mergesort T[4..4]
[5] done
[9] [5] merge
[5, 9] done
[2, 7] [5, 9] merge
[2, 5, 7, 9] done

T = [**2**, **5**, **7**, **9**, 1, 3, 8, 4] mergesort T[5..8]
[1, 3, 8, 4] mergesort T[5..6]
[1, 3] mergesort T[5..5]
[1] done
[**1**, 3] mergesort T[6..6]
[3] done
[1] [3] merge
[1, 3] done
[**1**, **3**, 8, 4] mergesort T[7..8]
[8, 4] mergesort T[7..7]
[8] done
[**8**, 4] mergesort T[8..8]
[4] done
[8] [4] merge
[4, 8] done
[1, 3] [4, 8] merge

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done
[**2**, **7**, 9, 5] mergesort T[3..4]
[9, 5] mergesort T[3..3]
[9] done
[**9**, 5] mergesort T[4..4]
[5] done
[9] [5] merge
[5, 9] done
[2, 7] [5, 9] merge
[2, 5, 7, 9] done

T = [**2**, **5**, **7**, **9**, 1, 3, 8, 4] mergesort T[5..8]
[1, 3, 8, 4] mergesort T[5..6]
[1, 3] mergesort T[5..5]
[1] done
[**1**, 3] mergesort T[6..6]
[3] done
[1] [3] merge
[1, 3] done
[**1**, **3**, 8, 4] mergesort T[7..8]
[8, 4] mergesort T[7..7]
[8] done
[**8**, 4] mergesort T[8..8]
[4] done
[8] [4] merge
[4, 8] done
[1, 3] [4, 8] merge
[1, 3, 4, 8] done

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done
[**2**, **7**, 9, 5] mergesort T[3..4]
[9, 5] mergesort T[3..3]
[9] done
[**9**, 5] mergesort T[4..4]
[5] done
[9] [5] merge
[5, 9] done
[2, 7] [5, 9] merge
[2, 5, 7, 9] done

T = [**2**, **5**, **7**, **9**, 1, 3, 8, 4] mergesort T[5..8]
[1, 3, 8, 4] mergesort T[5..6]
[1, 3] mergesort T[5..5]
[1] done
[**1**, 3] mergesort T[6..6]
[3] done
[1] [3] merge
[1, 3] done
[**1**, **3**, 8, 4] mergesort T[7..8]
[8, 4] mergesort T[7..7]
[8] done
[**8**, 4] mergesort T[8..8]
[4] done
[8] [4] merge
[4, 8] done
[1, 3] [4, 8] merge
[1, 3, 4, 8] done
[2, 5, 7, 9] [1, 3, 4, 8] merge

- Mergesort

T = [7, 2, 9, 5, 1, 3, 8, 4] mergesort T[1..4]
[7, 2, 9, 5] mergesort T[1..2]
[7, 2] mergesort T[1..1]
[7] done
[**7**, 2] mergesort T[2..2]
[2] done
[7] [2] merge
[2, 7] done
[**2**, **7**, 9, 5] mergesort T[3..4]
[9, 5] mergesort T[3..3]
[9] done
[**9**, 5] mergesort T[4..4]
[5] done
[9] [5] merge
[5, 9] done
[2, 7] [5, 9] merge
[2, 5, 7, 9] done

T = [**2**, **5**, **7**, **9**, 1, 3, 8, 4] mergesort T[5..8]
[1, 3, 8, 4] mergesort T[5..6]
[1, 3] mergesort T[5..5]
[1] done
[**1**, 3] mergesort T[6..6]
[3] done
[1] [3] merge
[1, 3] done
[**1**, **3**, 8, 4] mergesort T[7..8]
[8, 4] mergesort T[7..7]
[8] done
[**8**, 4] mergesort T[8..8]
[4] done
[8] [4] merge
[4, 8] done
[1, 3] [4, 8] merge
[1, 3, 4, 8] done
[2, 5, 7, 9] [1, 3, 4, 8] merge
[**1**, **2**, **3**, **4**, **5**, **7**, **8**, **9**] sorted

- Mergesort
  - Merge is in $\Theta(n)$
  - Merge is called $\Theta(\log n)$ times recursively
  - Mergesort is in $\Theta(n \log n)$
  - Note: mergesort uses an additional array X[1…n] (if X was local to merge, much more storage would be used because of recursive calls)

- Shell Sort

```
procedure shellsort(T[1..n])
   inc = n
   while inc > 1 do
      inc = inc ÷ 2
      for j = 1 to inc do
         k = j + inc
         while k ≤ n do
            done = false
            x = T[k]
            current = k; previous = current – inc
            while previous ≥ j and not done do
               if x < T[previous] then
                  T[current] = T[previous]
                  current = previous; previous = previous – inc
               else
                  done = true
            T[current] = x
            k = k + inc
```

- Shell sort

  T = [7, 2, 9, 5, 1, 3, 8, 4] inc = 4

- Shell sort
  - Analysis of Shell sort is difficult
  - Intervals $n \div 2$, $n \div 4$, .., 1 are not optimal but are easy to compute
  - With these intervals, worst case is in $\Theta(n^2)$
  - Better intervals are 1, 3, 7, .., $2^m - 1$ (Hibbard)
  - With these intervals, worst case is in $\Theta(n^{3/2})$

- Shell sort (Hibbard intervals)

  T = [7, 2, 9, 5, 1, 3, 8, 4] inc = 7

- Shell sort (Hibbard intervals)

  T = [7, 2, 9, 5, 1, 3, 8, 4] inc = 7

  [*7*, 2, 9, 5, 1, 3, 8, *4*] sort T[1], T[8] (insertion sort)

- Shell sort (Hibbard intervals)

  T = [7, 2, 9, 5, 1, 3, 8, 4] inc = 7

  [*7*, 2, 9, 5, 1, 3, 8, *4*] sort T[1], T[8] (insertion sort)

  [4, 2, 9, 5, 1, 3, 8, 7] inc = 3

- Shell sort (Hibbard intervals)

T = [7, 2, 9, 5, 1, 3, 8, 4] inc = 7

[*7*, 2, 9, 5, 1, 3, 8, *4*] sort T[1], T[8] (insertion sort)

[4, 2, 9, 5, 1, 3, 8, 7] inc = 3

[*4*, 2, 9, *5*, 1, 3, *8*, 7] sort T[1], T[4], T[7] (insertion sort)

Homework:

Implement   Heapsort