COMP2521 – Course Notes

Contents

Week 1

1.1: Introduction

- This course will get you to:
    - Think like a computer scientist
    - Know fundamental techniques/structures
    - Able to reason about applicability/effectiveness
    - Able to analyse behaviour/correctness of programs

1.2: COMP1511 Code Gap

COMP2521 Style

- Allowed to use all C control structures:
  - `if`
  - `switch`
  - `while`
  - `for`
  - `break`
  - `continue`
- Other constructs:
  - Assignment statements in expressions
  - Use conditional statements, but use: `x = c ? e1 : e2` (ternary operator) with care
  - Functions may have multiple return statements

Switch Statements

- This:

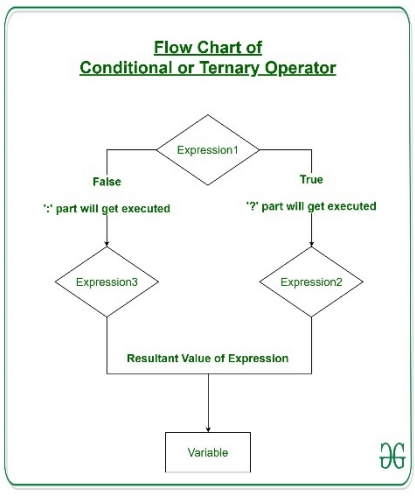
```
if (v == C1) {
    S1;
} else if (v == C2) {
    S2;
}
...
else if (v == Cn) {
    Sn;
}
else {
    Sn+1;
}
```

  can become this:

```
switch (v) {
case C1:
    S1; break;
case C2:
    S2; break;
...
case Cn:
    Sn; break;
default:
    Sn+1;
}
```

- Note: `break` is critical, otherwise it will fall through to the next case

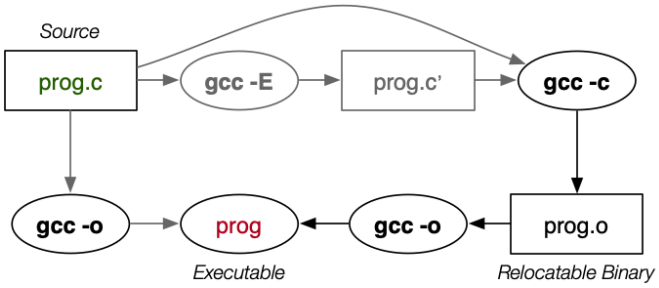
Ternary Operator



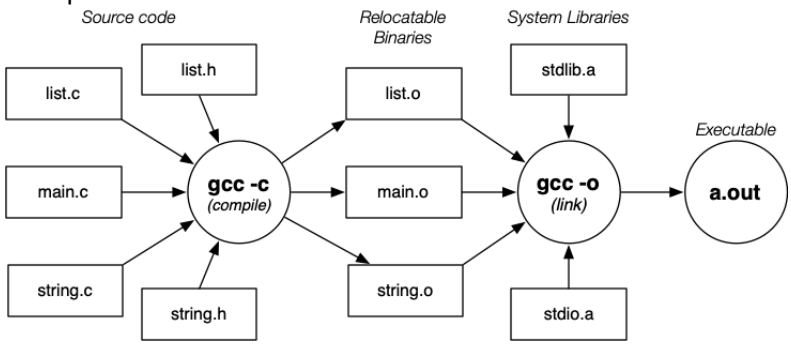
1.3: Compilation and Makefiles

Compilers

- Compilers are programs that:
  - convert program source code to executable form
  - 'executable' might be machine code or byte code
- The Gnu C Compiler (gcc)
  - Applies source-to-source transformation (pre-processor)
  - Compiles source code to produce object files
  - Links object files and libraries to produce executables
- `clang` is an alternative C compiler
- Stages in C compilation: pre-processing, compilation, linking



- When we have multiple C files involved:



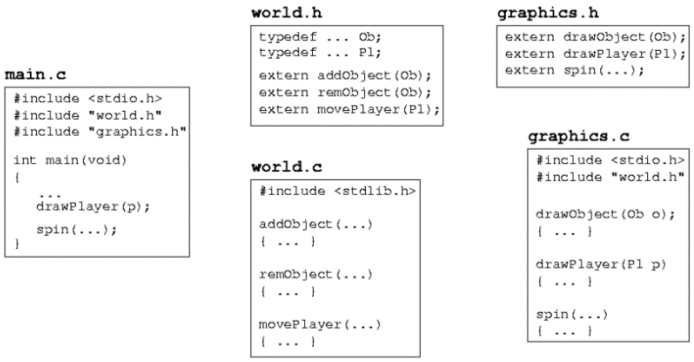
- Compilation/linking with `gcc`

```
gcc -c Stack.c
produces Stack.o, from Stack.c and Stack.h
gcc -c bracket.c
produces bracket.o, from bracket.c and Stack.h
gcc -o rbt bracket.o Stack.o
links bracket.o, Stack.o and libraries
producing executable program called rbt
```

- `gcc` is a multi-purpose tool, it compiles (`-c`), links, makes executables (`-o`)

Makefiles

- Compiling large systems is very complex
- The **make** command assists by allowing:
    - programmers to document dependencies in code
    - minimal re-compilation, based on dependencies
- For example:



- **make** is driven by dependencies given in the **Makefile**
- A dependency specifies:

```
target : source1 source2 …
         commands to build target from sources
```

for example:

```
game : main.o graphics.o world.o
       gcc -o game main.o graphics.o world.o
```

- Target is rebuilt if it is older than any other source
- For example:

```
game : main.o  graphics.o  world.o
       gcc -o game main.o graphics.o world.o

main.o : main.c graphics.h world.h
       gcc -Wall -Werror -c main.c

graphics.o : graphics.c world.h
       gcc -Wall -Werror -c graphics.c

world.o : world.c
       gcc -Wall -Werror -c world.c
```

- If `make` arguments are targets, build just those targets:

```
make world.o
gcc -Wall -Werror -c world.c
```

- If no arguments, build first target in the `Makefile`

```
make
gcc -Wall -Werror -c main.c
gcc -Wall -Werror -c graphics.c
gcc -Wall -Werror -c world.c
gcc -o game main.o graphics.o world.o
```

- Makefiles can also contain variables
    - e.g. **CC**, **CFLAGS**, **LDFLAGS**
    - can easily change which C compiler used, etc
- make has rules, which allow it to interpret e.g.

```
Stack.o : Stack.c Stack.h
```

As

```
Stack.o : Stack.c Stack.h
        $(CC) $(CFLAGS) -c Stack.c
```

1.4: Recursion (Linked List)

Recursion

- Recursion is a programming pattern where a function calls itself

Pattern for a Recursive Function

- Base case(s)
    - Situations when we do not call the same function (no recursive call), because the problem can be solved easily without recursion
    - All recursive cases eventually lead to one of the base cases
- Recursive case
    - We call the same function for a problem with smaller size
    - Decrease in a problem size eventually leads to one of the base cases
- Sum of list data fields

```
// return sum of list data fields: using recursive call

int sum(struct node *head) {
    if (head == NULL) {
        return 0;
    }
    return head->data + sum(head->next);
}
```

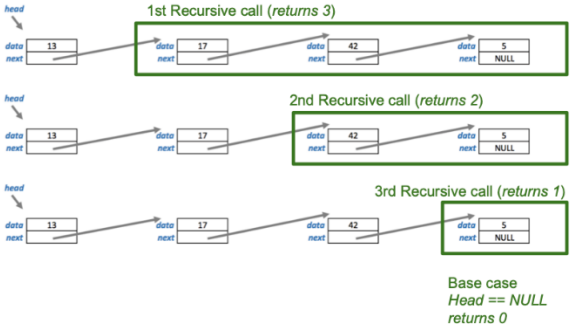
Base case

Recursive case, Recursive call for a smaller problem (size-1)

- Length of linked list

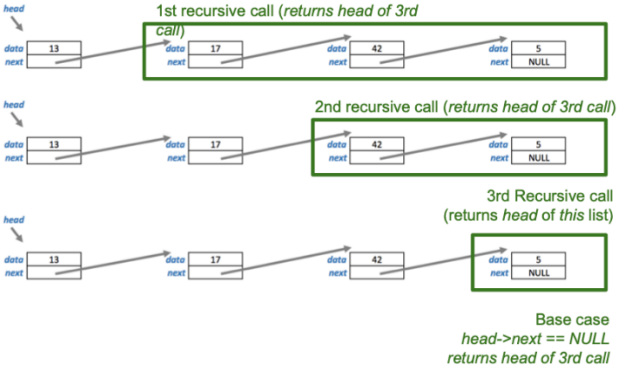
```
// return count of nodes in list

int length(struct node *head) {
    if (head == NULL) {
        return 0;
    }
    return 1 + length(head->next);
}
```

Recursive call



- Last node of a linked list

```
struct node *last(struct node *head) {
    // list is empty
    if(head == NULL) {
        return NULL;
    }
    // found the last node! return it.
    else if (head->next == NULL) {
        return head;
    }
    // return last node from the rest of the list
    // using a recursion
    else {
        return last(head->next);
    }

}
```



- Find node using recursion

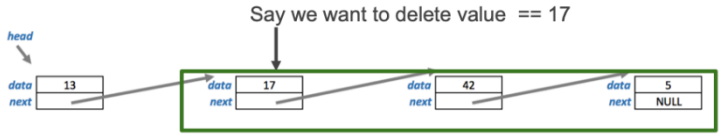
```
// return pointer to first node with specified data value
// return NULL if no such node

struct node *find_node(struct node *head, int data) {
    // empty list, so return NULL
    if (head == NULL) {
        return NULL;
    }
    // Data at "head" is same as the "data" we are searching,
    // Found the node! so return head.
    else if (head->data == data) {
        return head;
    }
    // Find "data" in the rest of the list, using recursion,
    // return whatever answer we get from the recursion
    else {
        return find_node(head->next, data);
    }

}
```

Recursive call

- Delete a node from a linked list

```
// Delete a Node from a List: Recursive
struct node *deleteR(struct node *list, int value) {
   if (list == NULL) {
      fprintf(stderr, "warning: value %d is not in list\n", value);

   } else if (list->data == value) {
      struct node *tmp = list;
      list = list->next;        // remove first item
      free(tmp);

   } else {
      list->next = deleteR(list->next, value);
   }
   return list;
}
```

Recursive call



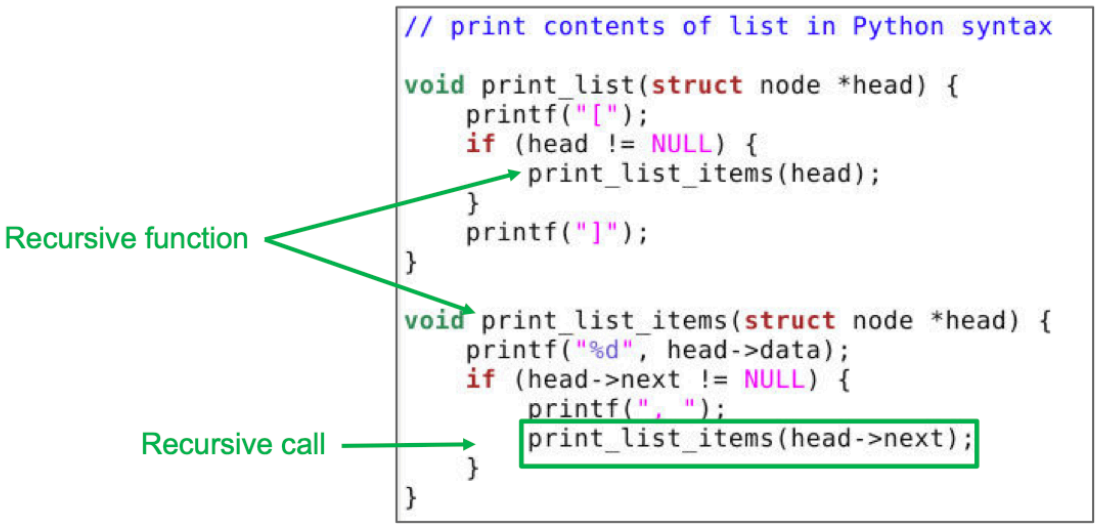
1st recursive call (node to delete is same as "head" of this call, *returns updated list, pointing to node with 42)*

- Insert a node to a linked list

```
// Insert a Node into an Ordered List: recursive
struct node *insertR(struct node *list, int value) {
   if (list == NULL || list->data >= value) {

      struct node *newHead = create_node(value, NULL);
      newHead->next = list;
      return newHead;

      // Alternatively, in one line
      // return create_node(value, list) ;
   }

   list->next = insertR(list->next, value);

   return list;
}
```

Recursive call

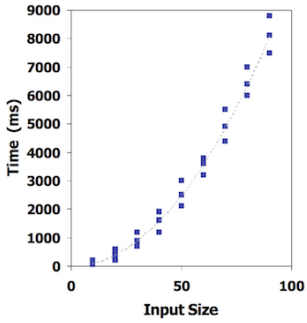
- Print Python List using Recursion



Week 2

2.1: Analysis of Algorithms

- An algorithm is a step-by-step procedure
  - For solving a problem
  - In a finite amount of time
- Most algorithms map input to output
  - Running time typically grows with input size
  - Average time is often difficult to determine
  - Focus on worst case running time



Empirical Analysis

- Write a program that implements an algorithm
- Run program with inputs of varying size and composition
- Measure actual running time
- Plot the results
- Limitations:
  - Requires implementing the algorithm, which may be difficult
  - Results may not be indicative of running time on other inputs
  - Same hardware and OS must be used to compare two algorithms

Theoretical Analysis

- Uses a high-level description of the code instead of implementation (pseudocode)
- Characterises running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent of the programming language
- Examples:
  - Evaluating an expression
  - Indexing into an array
  - Calling/returning from a function

Counting Primitive Operations

```
arrayMax(A):
|  Input  array A of n integers
|  Output maximum element of A
|
|  currentMax=A[0]                    1
|  for all i=1..n-1 do                n+(n-1)
|  |  if A[i]>currentMax then         2(n-1)
|  |     currentMax=A[i]              n-1
|  |  end if
|  end for
|  return currentMax                  1
                                       -------
                              Total    5n-2
```

- The time complexity is O(n)

```
matrixProduct(A,B):
|  Input  n×n matrices A, B
|  Output n×n matrix A·B
|
|  for all i=1..n do                      2n+1
|  |  for all j=1..n do                   n(2n+1)
|  |  |  C[i,j]=0                         n^2
|  |  |  for all k=1..n do                n^2(2n+1)
|  |  |     C[i,j]=C[i,j]+A[i,k]·B[k,j]   n^3·5
|  |  |  end for
|  |  end for
|  end for
|  return C                               1
                                          ------------
                                  Total   7n^3+4n^2+3n+2
```

- The order is $O(n^3)$

Big-Oh Notation

- Big-Oh notation gives an upper bound on the growth rate of a function
- Use big-Oh to rank functions according to their rate of growth

Big-Oh Rules

- If f(n) is a polynomial of degree d -> f(n) is $O(n^d)$
  - Lower-order terms are ignored
  - Constant factors are ignored

Asymptotic Analysis of Algorithms

- Asymptotic analysis of algorithms determines running time in big-Oh notation:
  - Find worst-case number of primitive operations as a function of input size
  - Express this function using big-Oh notation
- Example:
  - Algorithm arrayMax executes at most

```
prefixAverages1(X):
|  Input  array X of n integers
|  Output array A of prefix averages of X
|
|  for all i=0..n-1 do          O(n)
|  |  s=X[0]                    O(n)
|  |  for all j=1..i do         O(n^2)
|  |     s=s+X[j]               O(n^2)
|  |  end for
|  |  A[i]=s/(i+1)              O(n)
|  end for
|  return A                     O(1)
```

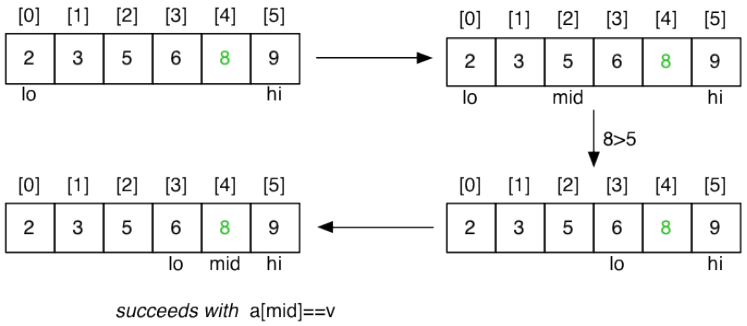
- The time complexity is $O(n^2)$

Binary Search

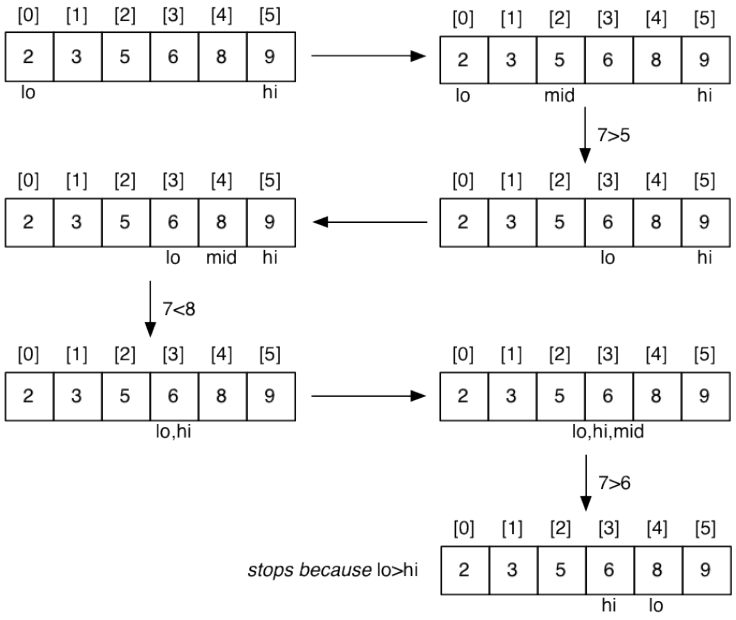
- Given a sorted array

```
search(v,a,lo,hi):
|  Input  value v
|         array a[lo..hi] of values
|  Output true if v in a[lo..hi]
|         false otherwise
|
|  mid=(lo+hi)/2
|  if lo>hi then return false
|  if a[mid]=v then
|     return true
|  else if a[mid]<v then
|     return search(v,a,mid+1,hi)
|  else
|     return search(v,a,lo,mid-1)
|  end if
```

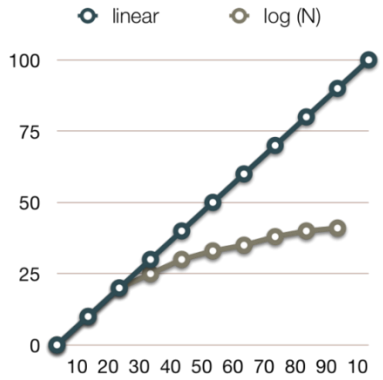
- Successful search for a value of 8:



- Unsuccessful search for a value of 7:



- Cost analysis:
  - C_i = #calls to **`search()`** for array of length i
  - for best case, $C_n = 1$
  - for **`a[i..j]`**, **`j<i`** (length=0)
    - $C_0 = 0$
  - for **`a[i..j]`**, **`i≤j`** (length=n)
    - $C_n = 1 + C_{n/2} \Rightarrow C_n = \log_2 n$
- Logarithmic complexity is good:



Exercises

```
splitList(L):
|  Input  non-empty linked list L
|  Output L split into two halves
|
|  // use slow and fast pointer to traverse L
|  slow=head(L), fast=head(L).next
|  while fast≠NULL ∧ fast.next≠NULL do
|     slow=slow.next, fast=fast.next.next  // advance pointers
|  end while
|  cut L between slow and slow.next
```

- Time complexity is O(|L|)

```
binaryConversion(n):
|  Input  positive integer n
|  Output binary representation of n on a stack
|
|  create empty stack S
|  while n>0 do
|  |  push (n mod 2) onto S
|  |  n=n/2
|  end while
|  return S
```

- Time complexity is O(log n)

Complexity Classes

- Problems in Computer Science
  - Some have polynomial worst-case performance
  - Some have exponential worst case performance
- Classes of problems:
  - P = problems for which an algorithm can compute answer in polynomial time
  - NP = includes problems for which no P algorithm is known
- Beware: NP stands for 'nondeterministic, polynomial time'

- Computer Science jargon for difficulty:
  - tractable: have a polynomial-time algorithm (useful in practice)
  - intractable: no tractable algorithm is known (feasible for only small n)
  - non-computable: no algorithm can exist

Generate and Test Algorithms

- In scenarios where
  - It is simple to test whether a given state is a solution
  - It is easy to generate new states (preferably likely solutions)
- Then a generate and test strategy can be used
- It is necessary that states are generated systematically
  - So that we are guaranteed to find a solution, or know that none exists
  - Some randomised algorithms do not require this

```
isPrime(n):
|  Input  natural number n
|  Output true if n prime, false otherwise
|
|  for all i=2..n-1 do       // generate
|  |  if n mod i = 0 then    // test
|  |     return false        // i is a divisor => n is not prime
|  |  end if
|  end for
|  return true               // no divisor => n is prime
```

- The complexity is O(n)

```
subsetsum1(A,k):
|  Input  set A of n integers, target sum k
|  Output true if Σ_{b∈B} b=k for some B⊆A
|         false otherwise
|
|  for s=0..2^n-1 do
|  |  if k = Σ_(i^th bit of s is 1) A[i] then
|  |     return true
|  |  end if
|  end for
|  return false
```

- The complexity is $O(2^n)$

```
subsetsum2(A,n,k):
|  Input  array A, index n, target sum k
|  Output true if some subset of A[0..n-1] sums up to k
|         false otherwise
|
|  if k=0 then
|     return true   // empty set solves this
|  else if n=0 then
|     return false  // no elements => no sums
|  else
|     return subsetsum2(A,n-1,k-A[n-1]) ∨ subsetsum2(A,n-1,k)
|  end if
```

- The complexity is $O(2^n)$
- The above two examples are NP-complete problems (intractable)

2.2: Abstract Data Types

- A data type is:
  - a set of values (atomic or structured values)
  - a collection of operations on those values
- An abstract data type is:
  - an approach to implementing data types
  - separates interface from implementation
  - users of the ADT see only the interface
  - builders of the ADT provide an implementation

DTs, ADTs, GADTs

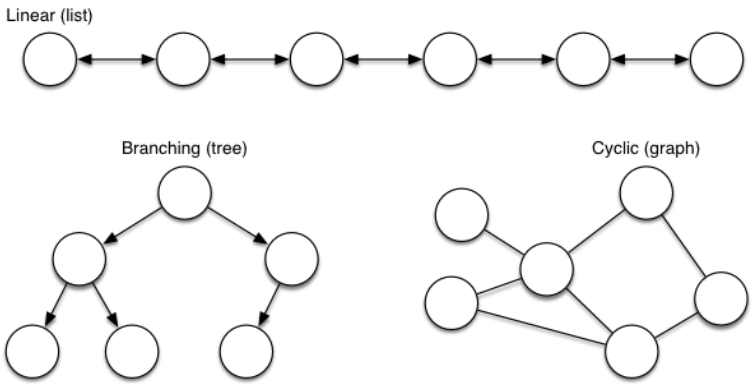
- We want to distinguish:
  - DT = (non-abstract) data type (e.g. C strings)
    - Internals of data structures are visible (e.g. char s[10];)
  - ADT = abstract data type (e.g. C files)
    - Can have multiple instances (e.g. Set a, b, c;)
  - GADT = generic (polymorphic) abstract data type
    - Can have multiple instances (e.g. Set <int> a, b, c;)
    - Can have multiple types (e.g. Set <int> a; Set <char> b;)
    - Not available natively in the C language

Interface/Implementation

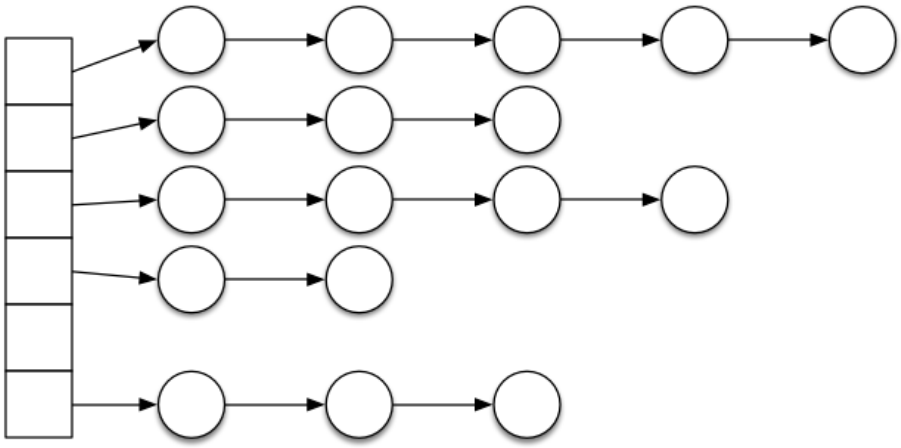
- ADT interface provides
  - A user-view of the data structure (e.g. FILE *)
  - Function signatures (prototypes) for all operations
  - Semantics of operations (via documentation)
  - A contract between ADT and its clients
- ADT implementation gives
  - Concrete definition of the data structures
  - Definition of functions for all operations

Collections

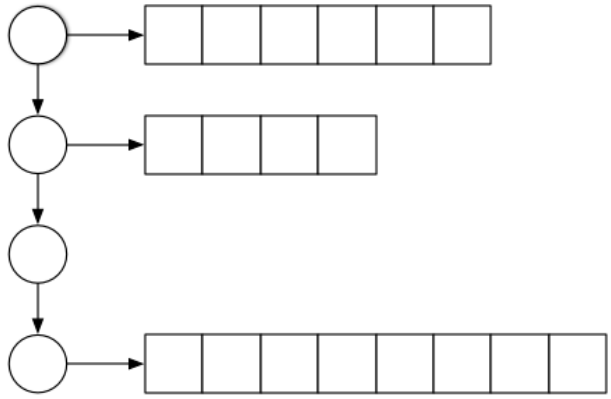
- Many of the ADTs we deal with
  - Consist of a collection of items
  - Where each item may be a simple type or an ADT
  - And items often have a key (to identify tem)
- Collections may be categorised by:
  - Structure: Linear (list), branching (tree), cyclic (graph)
  - Usage: Set, matrix, stack, queue, search-tree, dictionary



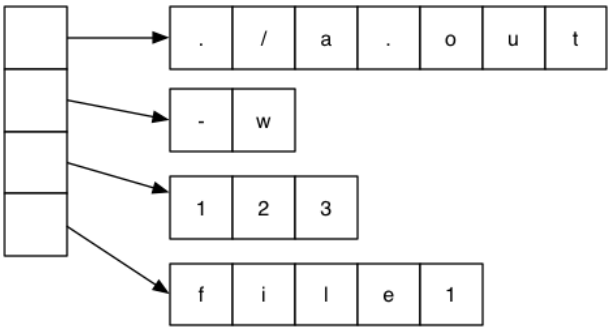
- Or even a hybrid structure:



Or this:



Or this:



- Typical operations on collections:
  - Create an empty collection
  - Insert one item into the collection
  - Remove one item from the collection
  - Find an item in the collection
  - Check properties of the collection (size, empty?)
  - Drop the entire collection
  - Display the collection

Week 3

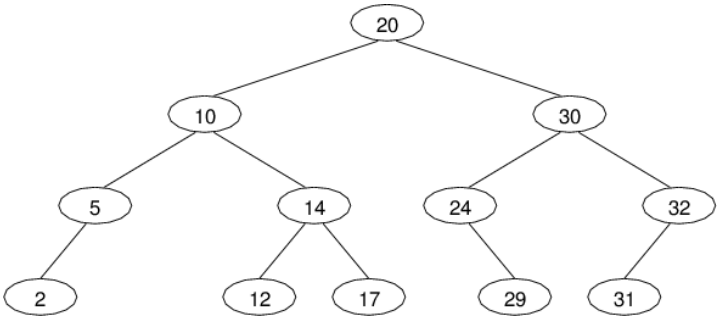
3.1: Tree, Search Trees

Searching

- Search is an extremely common application in computing
  - Given a (large) collection of items and a key value
  - Find the item(s) in the collection containing that key
    - item = (key, val_1, val_2, …) (i.e. a structured data type)
    - key = value used to distinguish items (e.g. student ID)
- Applications: Google, databases, …
- Many approaches have been developed for the 'search' problem
- Different approaches determined by properties of data structures:
  - Arrays: Linear, random-access, in-memory
  - Linked-lists: Linear, sequential access, in-memory
  - Files: Linear, sequential access, external
- Search costs:

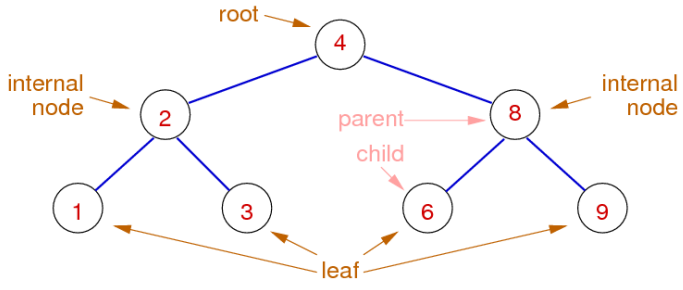
	Array	List	File
Unsorted	O(n) (linear scan)	O(n) (linear scan)	O(n) (linear scan)
Sorted	O(log n) (binary search)	O(n) (linear scan)	O(log n) (lseek,lseek,…)

- Maintaining arrays and files in sorted order is costly
- Search trees are efficient to search but also easy efficient to maintain
- For example, the following tree corresponds to the sorted array: [2,5,10,12,14,17,20,24,29,30,31,32]

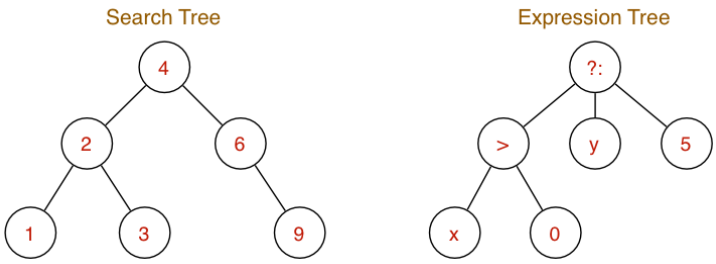


Tree Data Structures

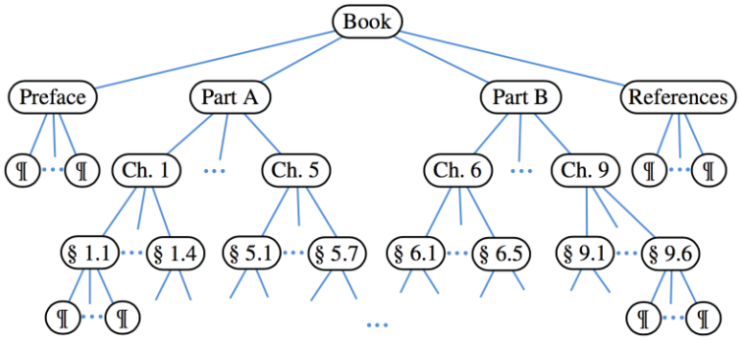
- Trees are connected graphs
  - With nodes and edges (called links), but no cycles (no 'up-links')
  - Each node contains a data value (or key+data)
  - Each node has links to <= 2 other child notes



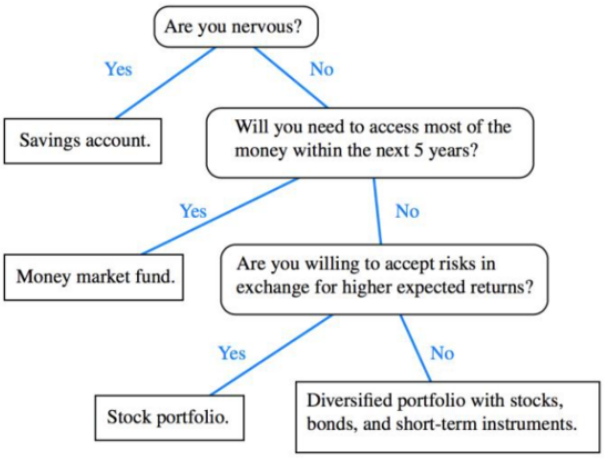
- Trees are used in many contexts, e.g.:
  - Representing hierarchal data structures (e.g. expressions)
  - Efficient searching (e.g. sets, symbol tables, …)



- Real word example:

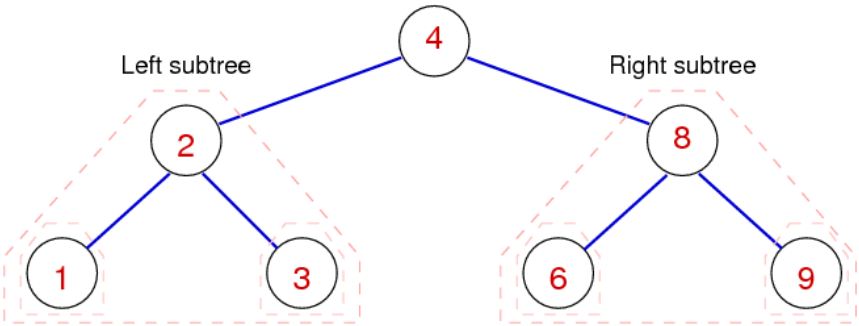


- Real world example: A decision tree



Binary Tree

- A binary tree is either:
  - Empty or
  - Consists of a node, with two subtrees
    - Node contains a value (typically key+data)
    - Left and right subtrees are binary trees (recursive)



Other Special Kinds of Tree

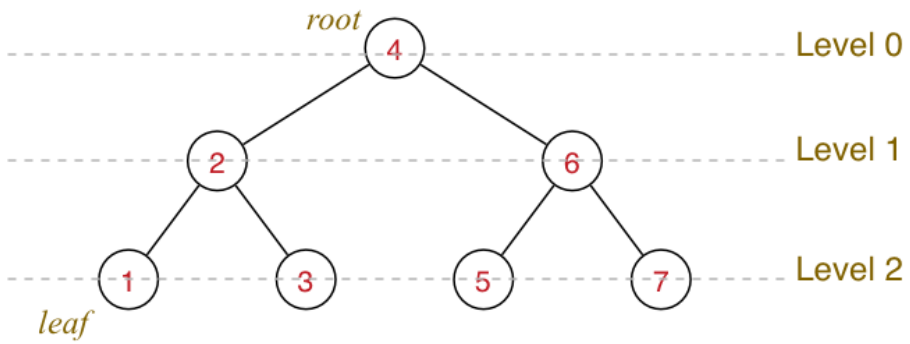
- *m*-ary tree: Each internal node has exactly *m* children
- B-tree: Each internal node has n/2 <= #children <= n
- Ordered tree: all left values < root, all right values > root
- Balanced tree: has minimal height for a given number of nodes
- Degenerate tree: has maximal height for a given number of nodes

Binary Search Tree

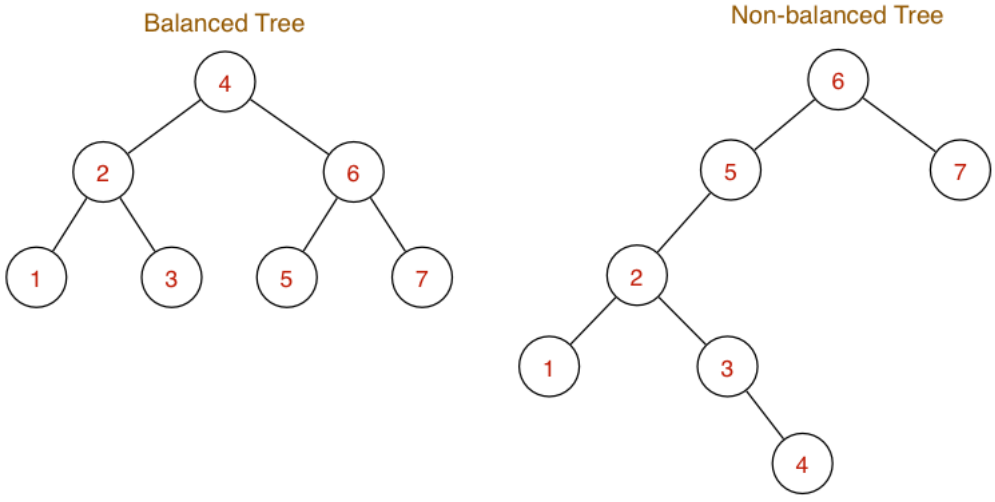
- Binary search trees (or BSTs) are ordered trees:
    - Each node is the root of 0, 1 or 2 subtrees
    - All values in any left subtree are less than root
    - All values in any right subtree are greater than root
    - These properties applies over all nodes in the trees

Properties of Binary Search Trees

- This tree has a height (or depth) of 2

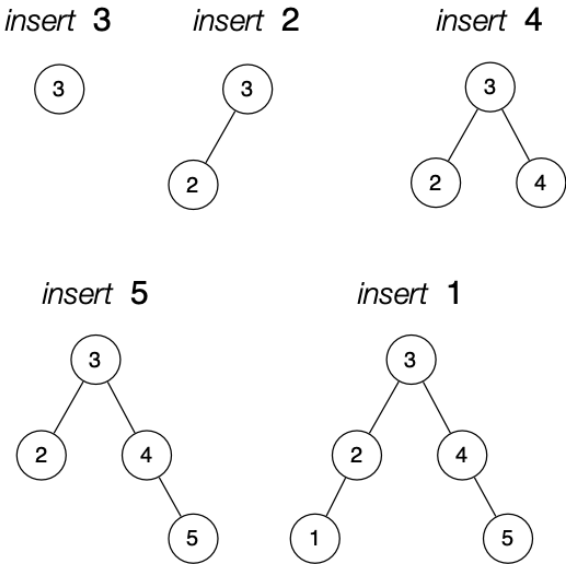


- Ordered tree: For all nodes, max(left subtree) < root < min(right subtree)
- Perfectly-balanced tree: For all nodes,
  |number_of_nodes(left subtree)| - |number_of_nodes(right subtree)| <= 1
- Height balanced tree: For all nodes,
- |height(left subtree)| - |height(right subtree)| <= 1
- Time complexity of a balanced tree algorithm is typically O(height)

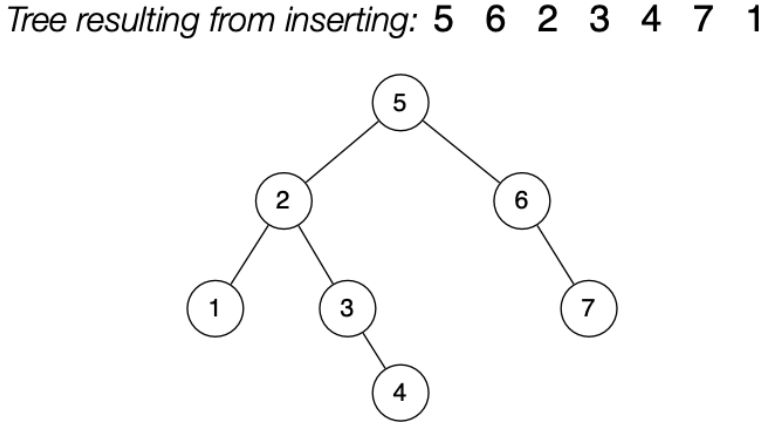


Insertion into BST

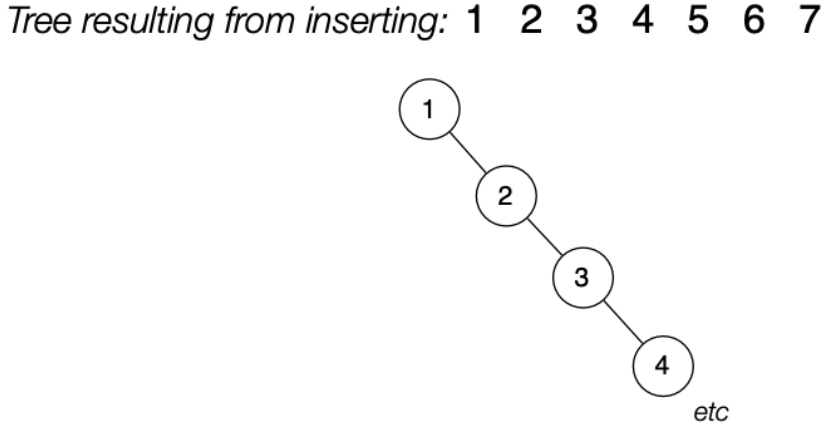
- Example 1:



- Example 2:

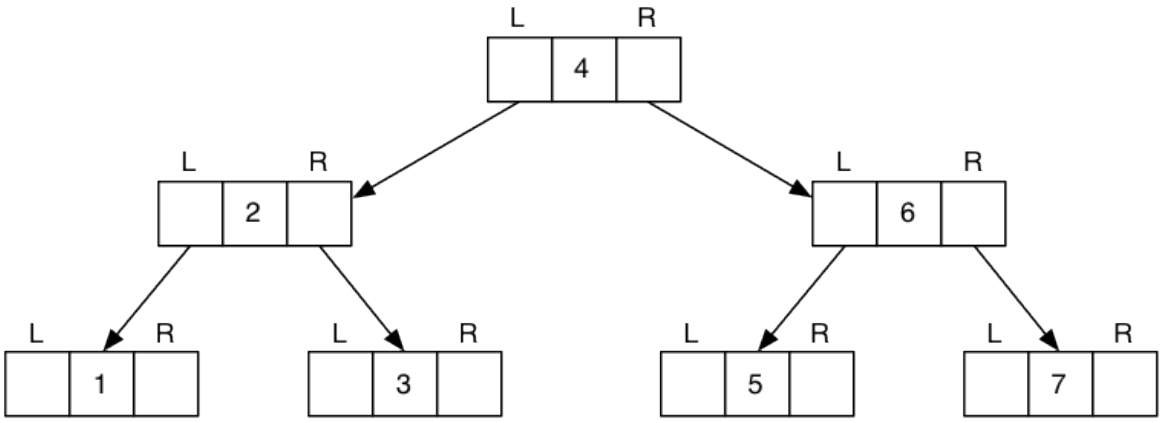


- Example 3:

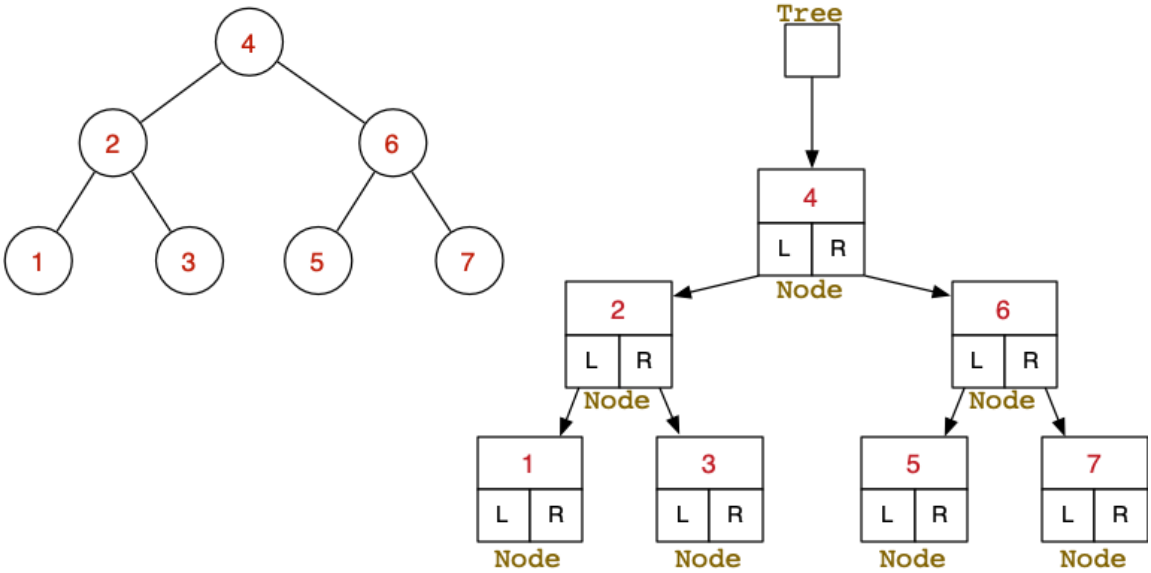


Representing BSTs

- Typically we use node structures




```
// a Tree is represented by a pointer to its root node
typedef struct Node *Tree;

// a Node contains its data, plus left and right subtrees
typedef struct Node {
   int  data;
   Tree left, right;
} Node;

// some macros that we will use frequently
#define data(node)  ((node)->data)
#define left(node)  ((node)->left)
#define right(node) ((node)->right)
```



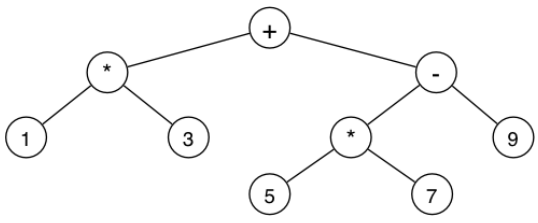
Tree Algorithms

```
TreeContains(tree,key):
|  Input  tree, key
|  Output true if key found in tree, false otherwise
|
|  if tree is empty then
|     return false
|  else if key < data(tree) then
|     return TreeContains(left(tree),key)
|  else if key > data(tree) then
|     return TreeContains(right(tree),key)
|  else          // found
|    return true
|  end if
```

```
TreeInsert(tree,item):
|  Input  tree, item
|  Output tree with item inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item < tree->data then
|     tree->left = TreeInsert(tree->left,item)
|     return tree
|  else if item > tree->data then
|     tree->right = TreeInsert(tree->right,item)
|     return tree
|  else
|     return tree     // avoid duplicates
|  end if
```

Tree Traversal

- Iteration (traversal on)
  - Lists: Visit each value from first to last
  - Graphs: Visit each vertex, order determined by DFS/BFS
- For binary trees, several well-defined visiting orders exist (L – left, R – right, N – root):
  - Preorder (NLR)
  - Inorder (LNR)
  - Postorder (LRN)
  - Level-order: visit root, then all its children, then all their children
- For example:



- NLR: + * 1 3 - * 5 7 9 (prefix-order: useful for building tree)
  LNR: 1 * 3 + 5 * 7 - 9 (infix-order: "natural" order)
  LRN: 1 3 * 5 7 * 9 - + (postfix-order: useful for evaluation)
  Level: + * - 1 3 * 9 5 7 (level-order: useful for printing tree)

- NLR (preorder traversal)

```
showBSTreePreorder(t):
```

```
|   Input tree t
|
|   if t is not empty then
|   |   print data(t)
|   |   showBSTreePreorder(left(t))
|   |   showBSTreePreorder(right(t))
|   end if
```

- LNR (inorder traversal)

```
showBSTreeInorder(t):
|   Input tree t
|
|   if t is not empty then
|   |   showBSTreePreorder(left(t))
|   |   print data(t)
|   |   showBSTreePreorder(right(t))
|   end if
```

- LRN (postorder traversal)

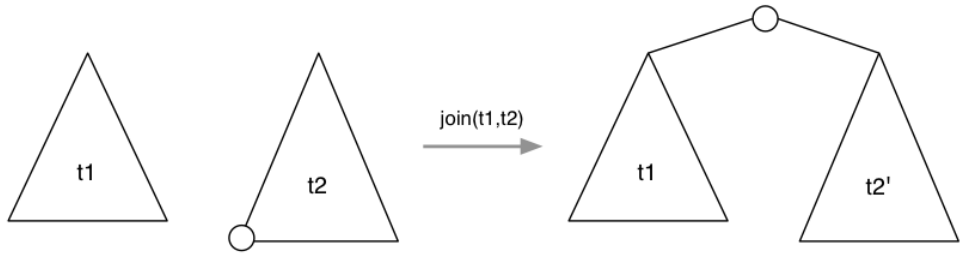
```
showBSTreePostorder(t):
|   Input tree t
|
|   if t is not empty then
|   |   showBSTreePreorder(left(t))
|   |   showBSTreePreorder(right(t))
|   |   print data(t)
|   end if
```

- Non-recursive NLR traversal

```
showBSTreePreorder(t):
|   Input tree t
|
|   push t onto new stack S
|   while stack is not empty do
|   |   t=pop(S)
|   |   print data(t)
|   |   if right(t) is not empty then
|   |      push right(t) onto S
|   |   end if
|   |   if left(t) is not empty then
|   |      push left(t) onto S
|   |   end if
|   end while
```

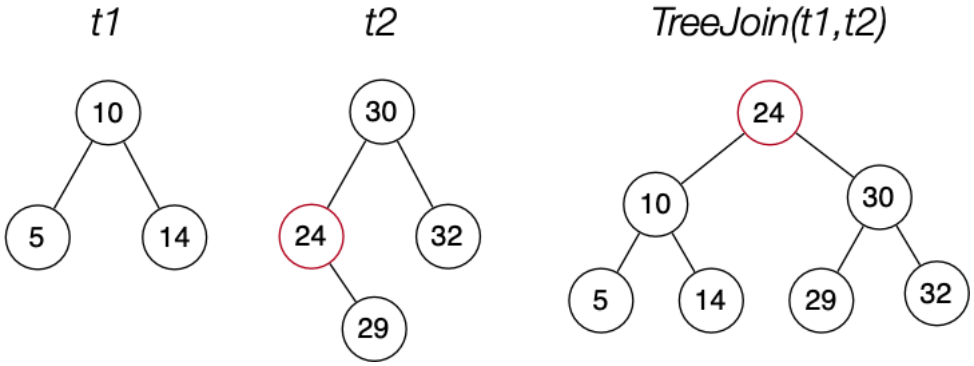
Joining Two Trees

- `t = TreeJoin(t1,t2)`
- Pre-conditions:
  - Takes two BSTs; returns a single BST
  - Max(key(t_1)) < Min(key(t_2))
- Post-conditions:
  - Result is a BST (i.e. fully ordered)
  - Containing all items from t_1 and t_2
- Method is to make the min node of the right subtree as the new root of both trees



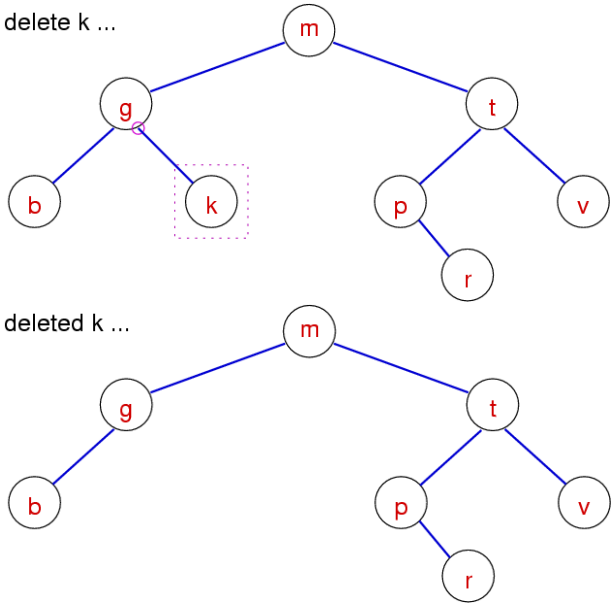

```
TreeJoin(t1,t2):
|  Input  trees t1,t2
|  Output t1 and t2 joined together
|
|  if t1 is empty then return t2
|  else if t2 is empty then return t1
|  else
|  |  curr=t2, parent=NULL
|  |  while left(curr) is not empty do     // find min element in t2
|  |     parent=curr
|  |     curr=left(curr)
|  |  end while
|  |  if parent≠NULL then
|  |     left(parent)=right(curr)  // unlink min element from parent
|  |     right(curr)=t2
|  |  end if
|  |  left(curr)=t1
|  |  return curr                  // curr is new root
|  end if
```

- Example:

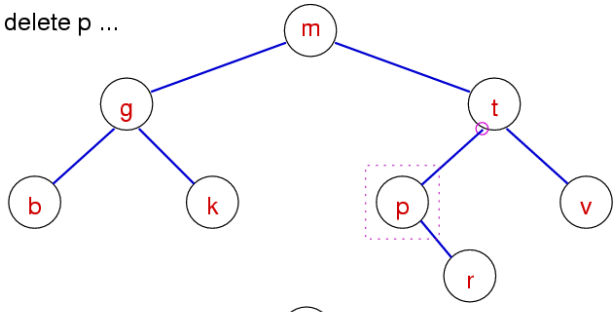


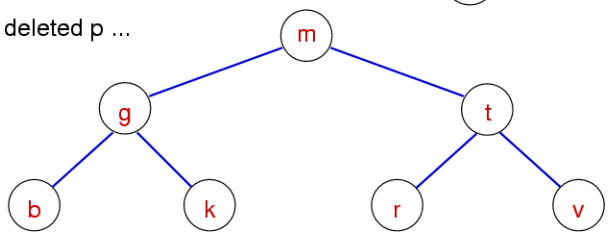
Deletion from BSTs

- Insertion into a binary search tree is easy
- Deletion from a binary search tree is harder
- Four cases to consider:
    - Empty tree, so new tree is also empty
    - Zero subtrees, so just delete the node

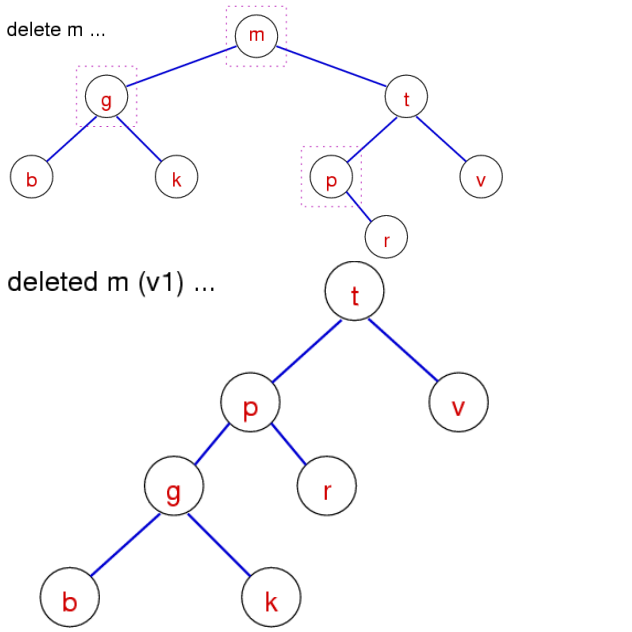


    - One subtree, so replace by child

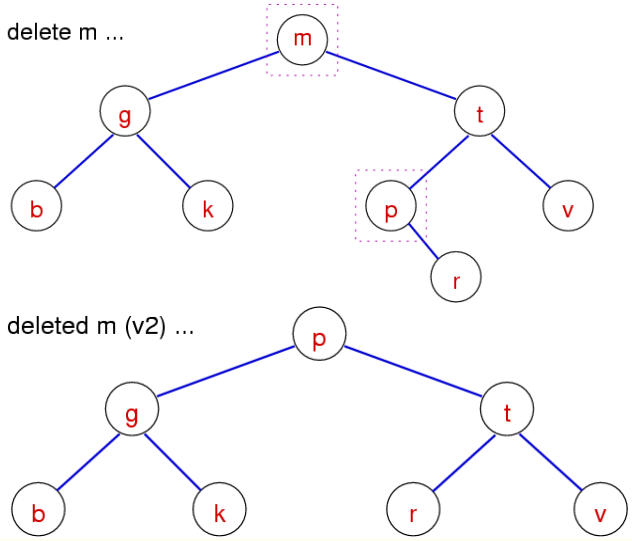




- Two subtrees, so replace by successor, join two subtrees
  - Version 1: right child comes new root, attach left subtree t min element of right subtree



  - Version 2: *Join* left and right subtree




```
TreeDelete(t,item):
|  Input  tree t, item
|  Output t with item deleted
|
|  if t is not empty then          // nothing to do if tree is empty
|  |  if item < data(t) then       // delete item in left subtree
|  |     left(t)=TreeDelete(left(t),item)
|  |  else if item > data(t) then  // delete item in left subtree
|  |     right(t)=TreeDelete(right(t),item)
|  |  else                        // node 't' must be deleted
|  |  |  if left(t) and right(t) are empty then
|  |  |     new=empty tree                  // 0 children
|  |  |  else if left(t) is empty then
|  |  |     new=right(t)                    // 1 child
|  |  |  else if right(t) is empty then
|  |  |     new=left(t)                     // 1 child
|  |  |  else
|  |  |     new=TreeJoin(left(t),right(t))  // 2 children
|  |  |  end if
|  |  |  free memory allocated for t
|  |  |  t=new
|  |  end if
|  end if
|  return t
```

3.2: Function Pointers in C

- C cab pass functions by passing a pointer to them
- Function pointers are
    - references to memory address of functions
    - pointer values and can be assigned/passed
- Function pointer variables/parameters are declared as:
  typeOfReturnValue (*fp)(typeOfArguments)
- Example: fp points to a function that returns int and have one argument of type int
  int (*fp) (int)
- Example 1:

```
void traverse (list ls, void (*fp) (list) ){
   list curr = ls;
   while(curr != NULL){
      // call function for the node
      fp(curr);
      curr = curr->next;
   }
}
```

Second argument is **fp**,
Pointer to a function like,
void functionName(list n)

```
void printNode(list n){
    if(n != NULL){
        printf("%d->",n->data);
    }
}
```

```
void printGrade(list n){
    if(n != NULL){
        if(n->data >= 50){
            printf("Pass");
        }
        else {
            printf("Fail");
        }
    }
}
```

```
void traverse (list ls, void (*fp) (list));


//The second argument must have matching prototype

traverse(myList,   printNode);
traverse(myList,   printGrade);
```

- Example 2:

```c
/* compare function
*/
int myCmp1(int val1, int val2){
    if(val1 < val2) { return -1;}
    else if (val1 > val2) { return 1; }
    else { return 0; }
}

/* Another compare function
*/
int myCmp2(int val1, int val2){
    if(val1 < val2) { return 1;}
    else if (val1 > val2) { return -1; }
    else { return 0; }
}
```

```c
/**
* Pre: n > 0, array a with at least one element
* Post: ??
 **/

int processA(int *a, int n, int (*fp) (int, int)){

    int cur = a[0];

        for(int i=1; i<n; i++){
        if( fp (cur, a[i]) < 0){
            cur = a[i];
        }
    }
    return cur;
}
```

```c
int main(int argc, char *argv[]) {

        int marksA[] = {67, 34, 81, 44, 19};

        int ans1 = processA (marksA, 5, myCmp1);
        printf("ans1 is %d \n", ans1);
        printf("------------------------------------ \n");

        int ans2 = processA (marksA, 5, myCmp2);
        printf("ans2 is %d \n", ans2);
        printf("------------------------------------ \n");

    return 0;
}
```

Week 4

4.1: Balancing Search Trees

- The goal is to build a binary search tree which has:
  - minimum height -> minimum worst case search cost

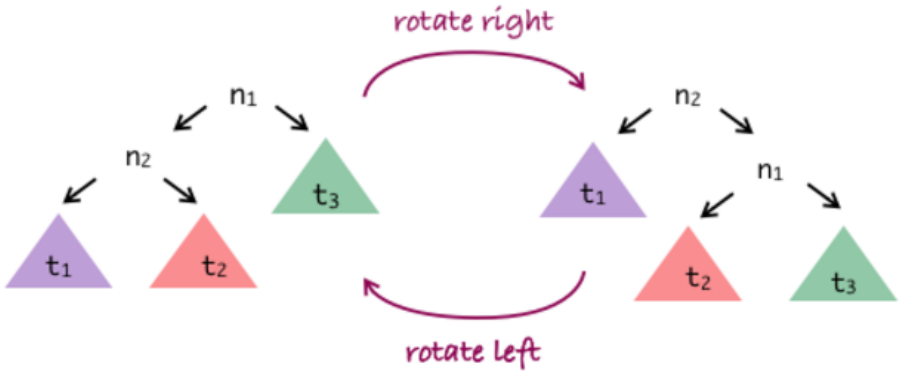
Perfectly Balanced Tree

- A perfectly balanced tree with N nodes has:
  - For all nodes,
    |number_of_nodes(left tree) – number_of_nodes(right tree)| < 2
  - Height of $\log_2 N$, so that the worst case search is $O(\log_2 N)$

Operations for Rebalancing

- Left rotation:
  - Move right child to root; rearrange links to retain order
- Right rotation
  - Move left child to root; rearrange links to retain order
- Insertion at root:
  - Each new item is added as the new root node
- Partition:
  - Rearrange tree around specified node (push it to root)

Tree Rotation

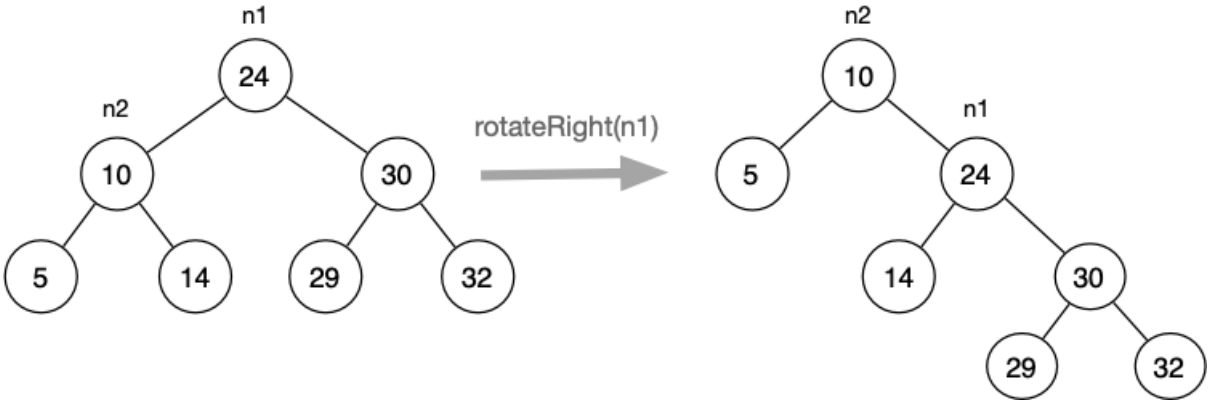


- Rotating right:
  - n_1 is the current root; n_2 is the root of n_1's left subtree
  - n_1 gets new left subtree, which is n_2's right subtree
  - n_1 becomes root of n_2's new right subtree
  - n_2 becomes new root
  - n_2's left subtree is unchanged
- Left rotation:
  - n_2 is the current root; n_1 is the root of n_2's right subtree
  - n_2 gets new right subtree, which is n_1's left subtree
  - n_2 becomes the root of n_1's new left subtree
  - n_1 becomes new root
  - n_1's right subtree is unchanged
- Rotation requires simple, localised pointer rearrangements
- Cost of tree rotation: O(1)

```
rotateRight(n1):
|   Input  tree n1
|   Output n1 rotated to the right
|
|   if n1 is empty V left(n1) is empty then
|      return n1
|   end if
|   n2= n1->left
|   n1->left = n2->right
|   n2->right =n1
|   return n2
```

```
rotateLeft(n2):
|   Input  tree n2
|   Output n2 rotated to the left
|
|   if n2 is empty V right(n2) is empty then
|      return n2
|   end if
|   n1=n2->right
|   n2->right = n1->left
|   n1->left = n2
|   return n1
```

- Example:



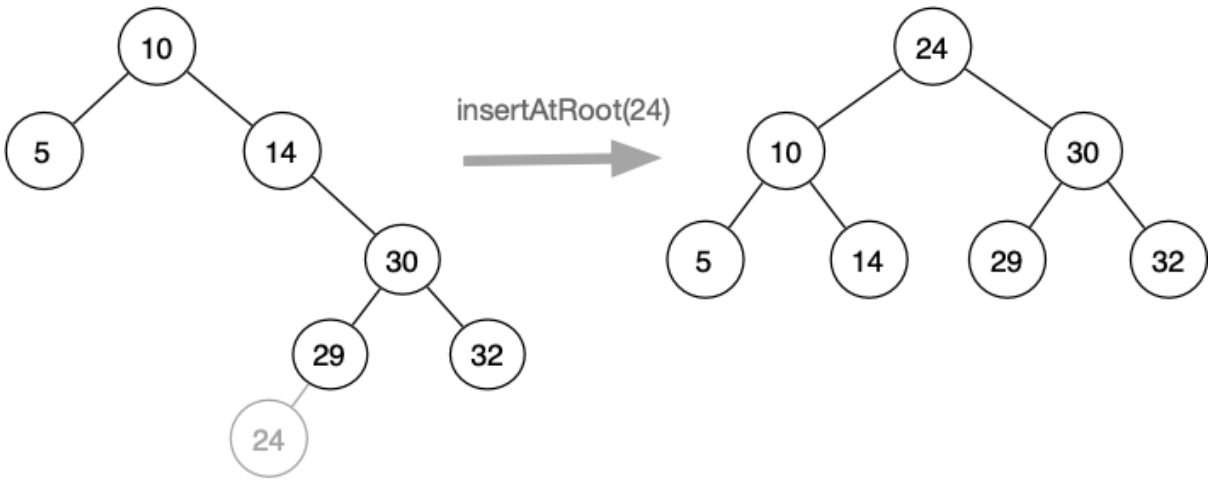
- Tree rotation is cheap: O(1)
- If applied appropriately, will tend to improve tree balance
- Sometimes rotation is applied from leaf to root, along one branch
    - Cost of this is O(height)
    - Payoff is improved balance which reduces height
- Reduced height pushes search cost to O(logN)

Insertion at Root

- Potential disadvantages:
  - Large-scale rearrangement of tree for each insert
- Potential advantages:
  - Recently inserted items are close to root
  - Lower cost if recent items more likely to be searched
- Method:
  - Base case:
    - Tree is empty; make new node and make it root
  - Recursive case:
    - Insert new node as root of appropriate subtree
    - Lift new node to root by rotation

```
insertAtRoot(t, it):
|   Input tree t, item it to be inserted
|   Output modified tree with item at root
|
|   if t is empty tree then
|      t = new node containing item
|   else if item < t->data then
|      t->left = insertAtRoot( t->left, it)
|      t = rotateRight(t)
|   else if it > root(t) then
|      t->right = insertAtRoot( t->right, it)
|      t = rotateLeft(t)
|   end if
|   return t;
```

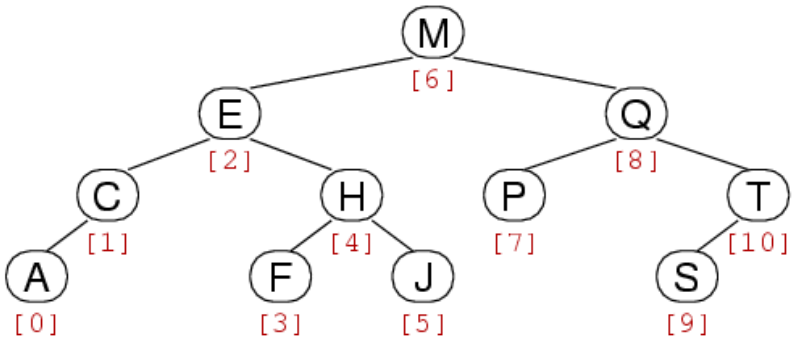
- Example:



- Same complexity as for insertion-at-leaf: O(height)
  - But cost is effectively doubled … traverse down, rotate up
  - Tendency to be balanced, but not guaranteed
- Benefit comes in searching
  - For some applications, search favours recently-added items
  - Insertion-at-root ensures these are close to root
- Could even consider ‘move to root when found’

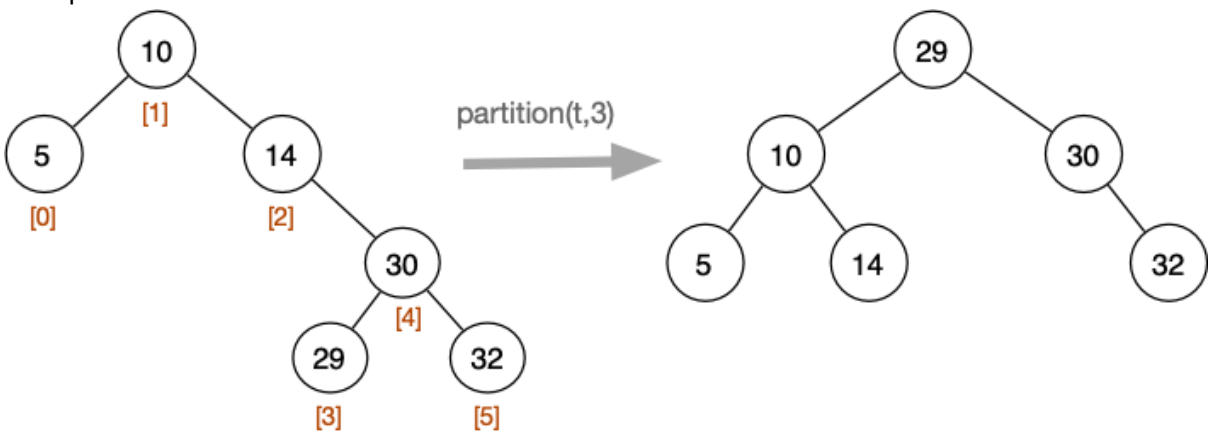
Tree Partitioning

- Rearranges so that element with index *i* becomes root
- For three with N nodes, indices are 0 .. N-1, in LNR order




```
partition(tree,i):
|   Input  tree with n nodes, index i
|   Output tree with i^th item moved to the root
|
|   m = number_of_nodes(left(tree))
|   if i < m then
|      tree->left = partition(tree->left, i)
|      tree = rotateRight(tree)
|   else if i > m then
|      tree->right = partition(tree->right , i-m-1)
|      tree = rotateLeft(tree)
|   end if
|   return tree
```

- Example:



- No requirement for search (using element index instead)
- After each recursive portioning step, one rotation
- Overall cost similar to insert-at-root
- Benefits:
    - Improves balance, so improves search cost

Periodic Rebalancing

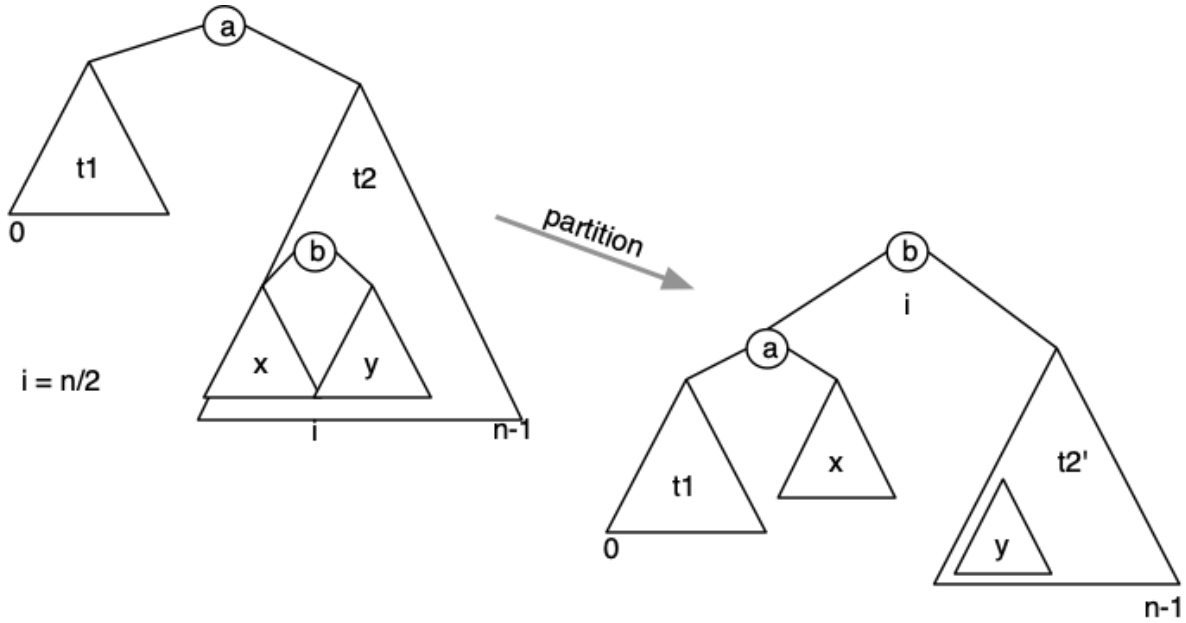
- Insert at leaves as before; periodically, rebalance the tree every k insertions

```
|  Input  tree, item
|  Output tree with item randomly inserted
|
|  t=insertAtLeaf(tree,item)
|  if #nodes(t) mod k = 0 then
|     t=rebalance(t)
|  end if
|  return t
```

- Problems:
  - Has to traverse entire subtree to count number of nodes
- Better to change nodes structure

```
typedef struct Node {
   int  data;
   int  nnodes;       // #nodes in my tree
   Tree left, right; // subtrees
} Node;
```

- How to rebalance a BST? Move median item to root




```
rebalance(t):
|  Input  tree t with n nodes
|  Output t rebalanced
|
|  if n≥3 then
|  |  // put node with median key at root
|  |  t=partition(t, n/2)
|  |  // then rebalance each subtree
|  |  t->left = rebalance(t->left)
|  |  t->right = rebalance(t->right)
|  end if
|  return t
```

Randomised BST Insertion

```
insertRandom(tree,item)
|   Input  tree, item
|   Output tree with item randomly inserted
|
|   if tree is empty then
|      return new node containing item
|   end if
|   // p/q chance of doing root insert
|   if random() mod q < p then
|      return insertAtRoot(tree,item)
|   else
|      return insertAtLeaf(tree,item)
|   end if
```

- E.g. 30% chance, choose p = 3, q = 10
- Similar to cost for inserting keys in random order: O(log N)
- Does not rely on keys being supplied in random order
- Approach can also be applied to deletion:
    - Promote inorder successor from right subtree OR left subtree

4.2: AVL Trees

Better Balanced Binary Search Trees

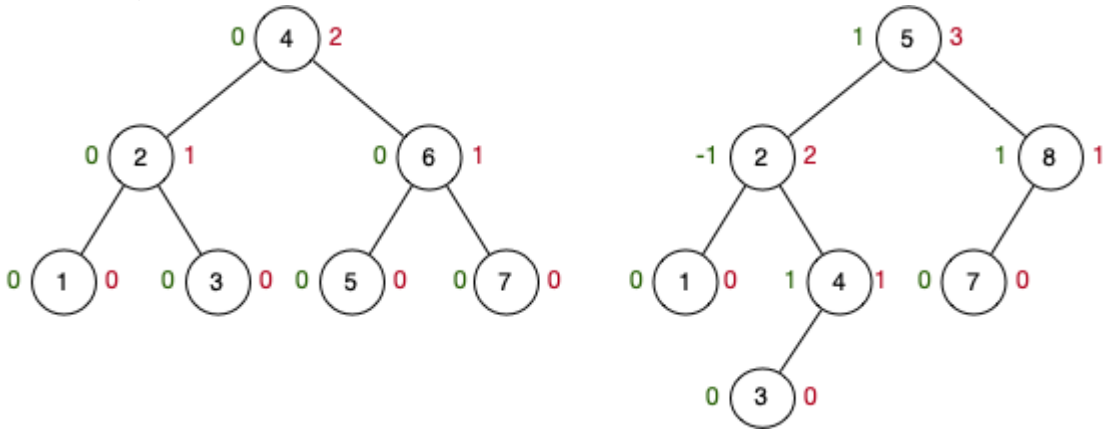
- So far, we have seen …
  - Randomised trees … make poor performance unlikely
  - Occasional rebalance … fix balance periodically
  - Splay trees … reasonable amortized performance
  - But all types still have O(n) worst case
- Ideally we want both average/worst case to be O(log N)
  - AVL trees … fix imbalances as soon as they occur
  - 2-3-4 trees … use varying sized nodes to assist balance
  - Red-black trees … isomorphic to 2-3-4, but binary nodes

AVL Trees

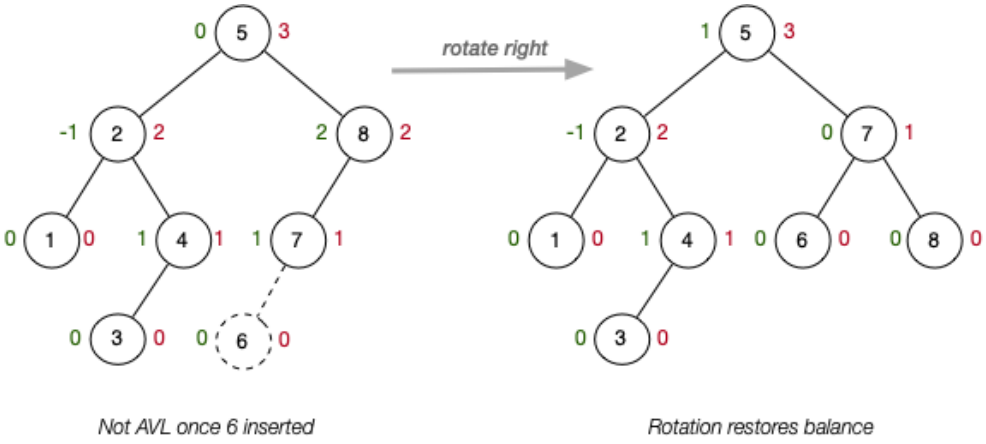
- Goal:
  - Tree remains reasonably well-balanced O(log N)
  - Cost of fixing imbalance is relatively cheap
- Approach:
  - Insertion (at leaves) may cause imbalance
  - Repair balance as soon as we notice imbalance
  - Repairs done locally, not by overall tree restructure

- A tree is unbalanced when |height(left subtree) – height (right subtree)| > 1
- This can be repaired by rotation:
  - If left subtree too deep, rotate right
  - If right subtree too deep, rotate left
- Problem: Determining height/depth of subtrees is expensive
  - Need to traverse whole subtree to find longest path
- Solution: Store balance data in each node

- Real numbers are height, green numbers are balance (difference in height between left and right subtree)



- Example of AVL tree rebalancing:



AVL Insertion

```
insertAVL(tree,item):
|  Input  tree, item
|  Output tree with item AVL-inserted
|
|  if tree is empty then
|     return new node containing item
|  else if item = data(tree) then
|     return tree
|  else
|  |  if item < data(tree) then
|  |     left(tree) = insertAVL(left(tree),item)
|  |  else if item > data(tree) then
|  |     right(tree) = insertAVL(right(tree),item)
|  |  end if
|  |  LHeight = height(left(tree))
|  |  RHeight = height(right(tree))
|  |  if (LHeight - RHeight) > 1 then
|  |     if item > data(left(tree)) then
|  |        left(tree) = rotateLeft(left(tree))
|  |     end if
|  |     tree=rotateRight(tree)
|  |  else if (RHeight - LHeight) > 1 then
|  |     if item < data(right(tree)) then
|  |        right(tree) = rotateRight(right(tree))
|  |     end if
|  |     tree=rotateLeft(tree)
|  |  end if
|  |  return tree
|  end if
```

Performance of AVL Trees

- Analysis of AVL trees:
  - Trees are height-balanced; subtree depts differ by +/- 1
  - Average/worst-case search performance of O(log N)
  - Require extra data to be stored in each node (efficiency)
  - Require extra data to be maintained during insertion
  - May not be weight-balanced; subtree sizes may differ

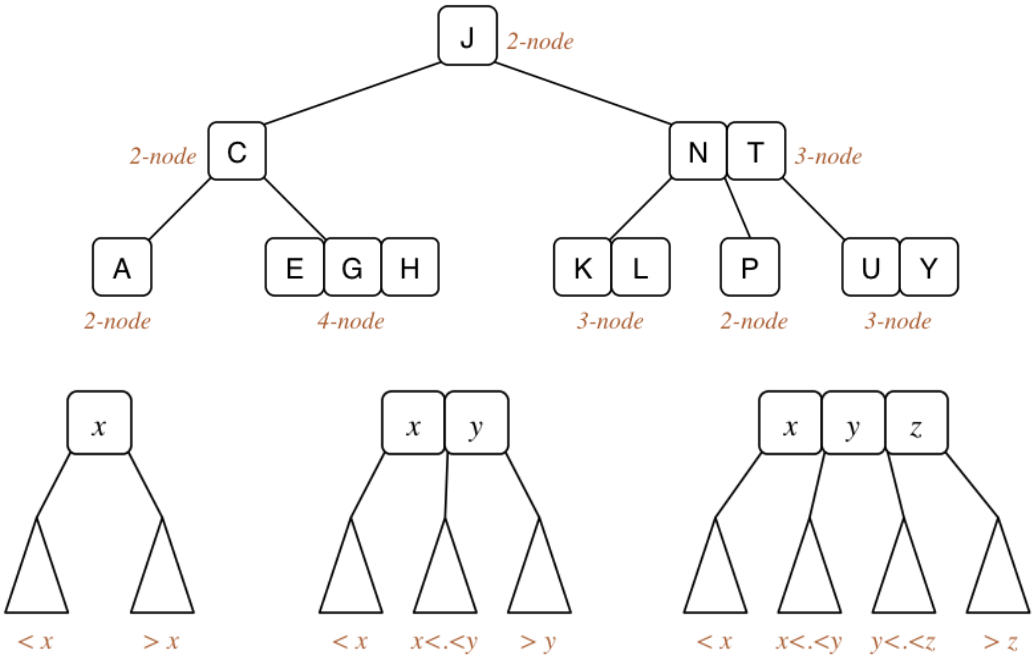
4.3: 2-3-4 Trees

Search Cost

- Critical factor determining search cost in BSTs
    - Worst case: Length of longest path
    - Average case: < average path length
- Either way, path length (tree depth) is a critical factor
- What if branching factor is > 2?
- $\log_2 4096 = 12$, $\log_4 4096 = 6$, $\log_8 4096 = 4$

2-3-4 Trees

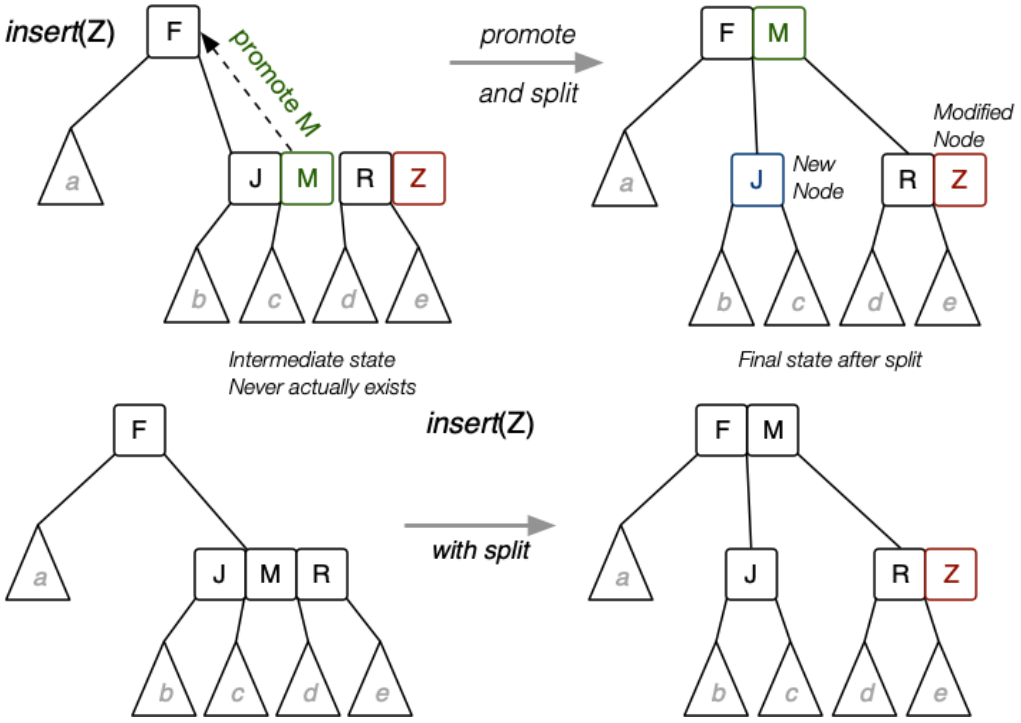
- 2-nodes, with two children (same as BSTs)
- 3-nodes, two values, three children
- 4-nodes, three values and four children



- In a balanced 2-3-4 tree:
    - All leaves are at same distance from the root
- 2-3-4 trees grow ‘upwards’ from the leaves, via node splitting

Node Splitting

- Insertion into a full node causes a split
- The middle value is propagated to the parent node
- Values in the original node split across original node and new node

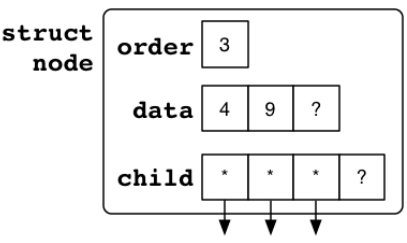


Searching in 2-3-4 Trees:

```
Search(tree,item):
|  Input  tree, item
|  Output address of item if found in 2-3-4 tree
|         NULL otherwise
|
|  if tree is empty then
|     return NULL
|  else
|  |  scan tree.data to find i such that
|  |     tree.data[i-1] < item ≤ tree.data[i]
|  |  if item=tree.data[i] then   // item found
|  |     return address of tree.data[i]
|  |  else       // keep looking in relevant subtree
|  |     return Search(tree.child[i],item)
|  |  end if
|  end if
```

Possible Data Structure

```
typedef struct node {
   int          order;     // 2, 3 or 4
   int          data[3];   // items in node
   struct node *child[4];  // links to subtrees
} node;
```

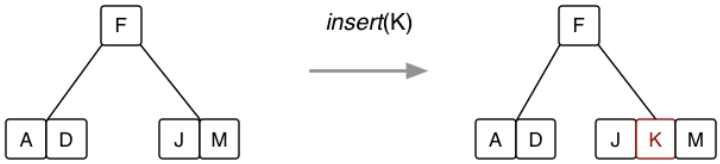


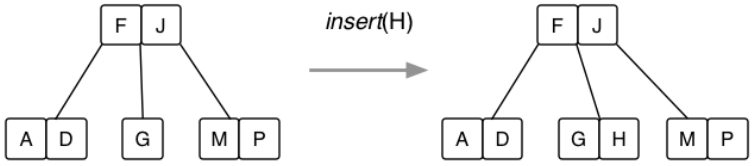
Search Cost Analysis

- In other trees, worst case is determined by height
- 2-3-4 trees are always balanced, so height is O(log N)
- Worst case for height: All nodes are 2-nodes: O(log N)
- Best case for height: All nodes are 4-nodes: $O(\log_4 N)$

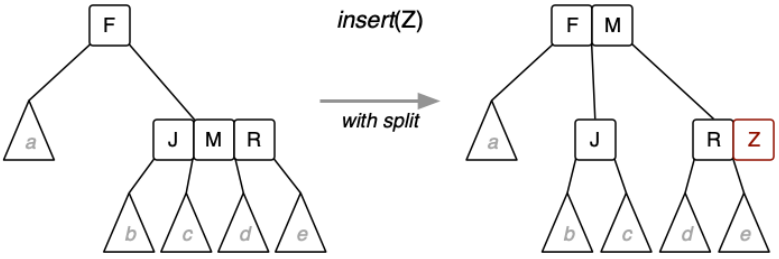
Insertion into 2-3-4 Trees

- Find a leaf node where the item belongs
- If not full (i.e. order < 4)
    - Insert item in this node, order++
- If node is full (i.e. contains 3 items)
    - Split into two 2-nodes as leaves
    - Promote middle element to parent
    - Insert item into appropriate leaf 2-node
    - If parent is a 4-node
        - Continue split/promote upwards
    - If promote to root, and root is a 4-node
        - Split root node and add new root
- Example: Insertion into a 2-node or 3-node:

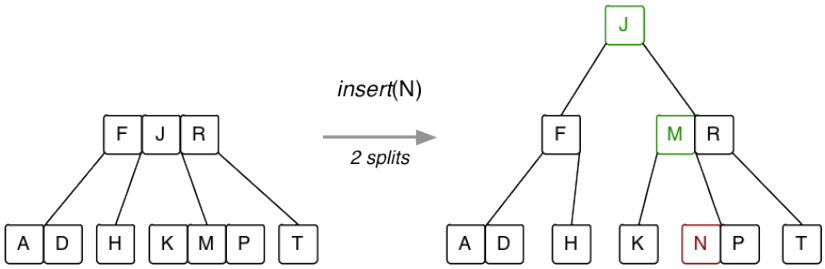




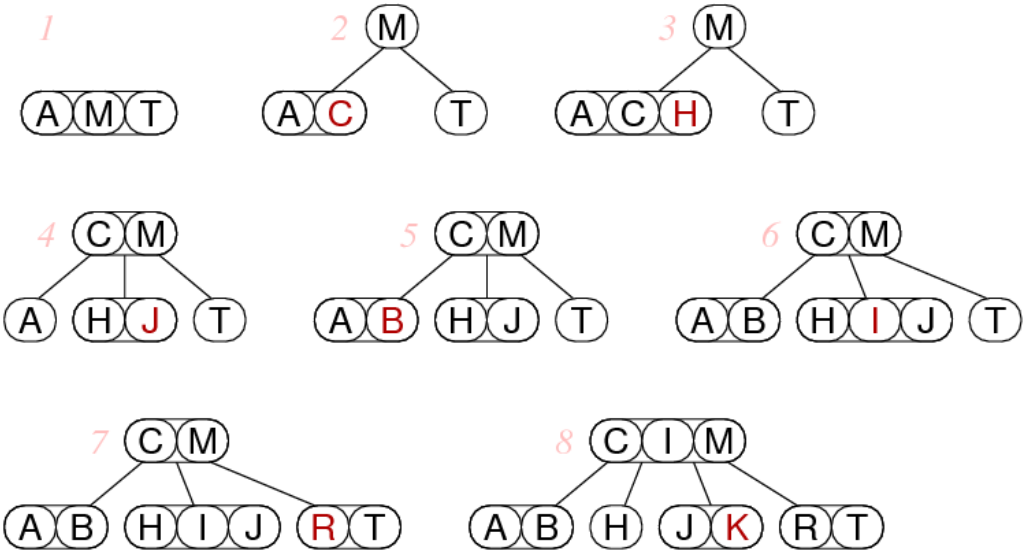
- Example: Insertion into 4-node (requires a split)



- Example: Splitting the root



- Example:



Insertion Algorithm

```
insert(tree,item):
|   Input  2-3-4 tree, item
|   Output tree with item inserted
|
|   if tree is empty then
|      return new node containing item
|   end if
|   node=Search(tree,item)
|   parent=parent of node
|   if node.order < 4 then
|      insert item into node
|      increment node.order
|   else
|   |   promote = node.data[1]      // middle value
|   |   nodeL    = new node containing data[0]
|   |   nodeR    = new node containing data[2]
|   |   delete node
|   |   if item < promote then
|   |      insert(nodeL,item)
|   |   else
|   |      insert(nodeR,item)
|   |   end if
|   |   insert(parent,promote)
|   |   while parent.order=4 do
|   |      continue promote/split upwards
|   |   end while
|   |   if parent is root ∧ parent.order=4 then
|   |      split root, making new root
|   |   end if
|   end if
```

Performance Analysis

- Searching for leaf node in which to insert = O(log N)
- If node not full, insert item into node = O(1)
- If node fill, promote middle, create two new nodes = O(1)
- If promotion propogates…
    - Best case: Update parent = O(1)
    - Worst case: Propagate to root = O(log N)
- Overall insertion cost = O(log N)

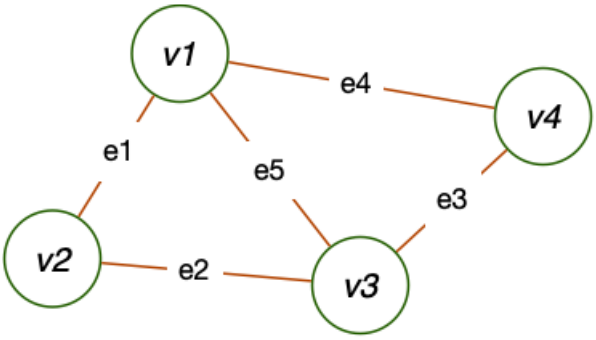
2-3-4 Variations

- Why stop at 4? Why no 2-3-4-5 trees? Or M-way trees?
- Or don't have variable sized nodes, simulate 2-3-4 trees with binary trees -> red black trees

Week 5

5.1: Graph Basics

- A graph G = (V, E):
    - V is a set of vertices
    - E is a set of edges (subset of VxV)



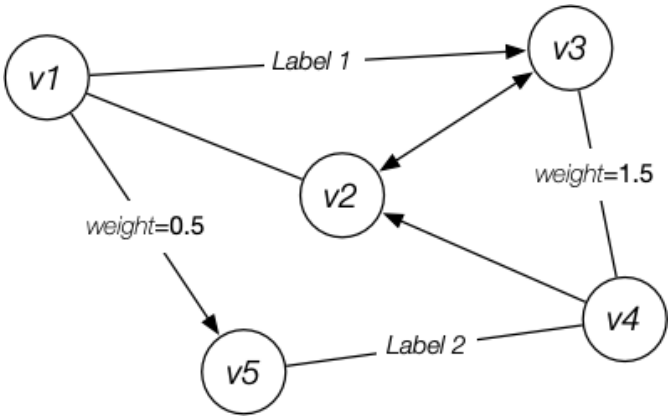
V = { *v1*, *v2*, *v3*, *v4* }

E = { e1, e2, e3, e4, e5 }

or

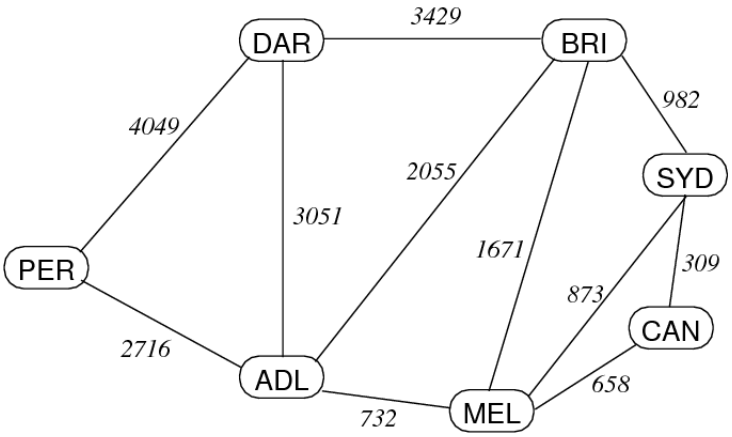
E = { (*v1*,*v2*), (*v2*,*v3*), (*v3*,*v4*), (*v1*,*v4*), (*v1*,*v3*) }

- Nodes are distinguished by a unique identifier
- Edges may be directed, labelled and/or weighted



- Example:

Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	1390	3051	732	2716	1605
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-



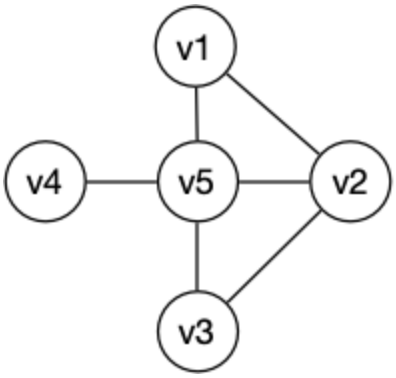
- Graph algorithms are generally more complex than tree/list ones:
    - No implicit order of items
    - Graphs may contain cycles
    - Concentre representation is less obvious
    - Algorithm complexity depends on connection complexity

Properties of Graphs

- A graph with V vertices has maximum V(V-1)/2 edges
- The ratio E:V can vary
    - If E is closer to V^2, the graph is dense
    - If E is closer to V, the graph is sparse
- Knowing if the graph is sparse or dense is important
    - May affect choice of data structures to represent the graph
    - May affect choice of algorithms to process graph

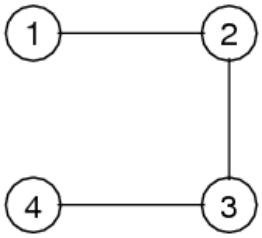
Graph Terminology

- For an edge *e* that connects vertices *v* and *w*
    - *v* and *w* are adjacent (neighbours)
    - *e* is incident on both *v* and *w*
- Degree of a vertex *v*
    - Number of edges incident on *e* (how many edges the vertex *v* has)

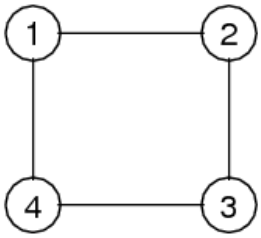


degree(v1) = 2
degree(v2) = 3
degree(v3) = 2
degree(v4) = 1
degree(v5) = 4

- Synonyms:
    - Vertex = Node
    - Edge = Arc = Link (note: some people use arc for directed edges)
- Path: A sequence of vertices where each vertex has an edge to its predecessor
- Cycle: A path where last vertex in path is same as first vertex in path
- Length of path or cycle = #edges

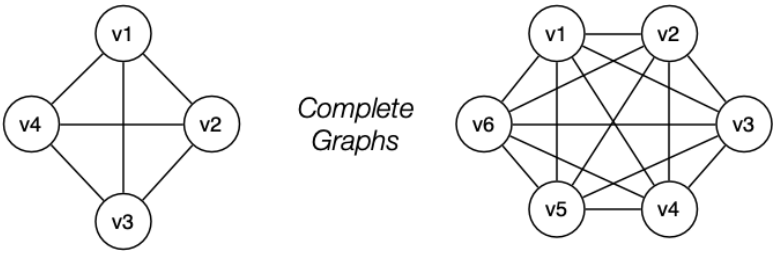


Path: 1–2, 2–3, 3–4

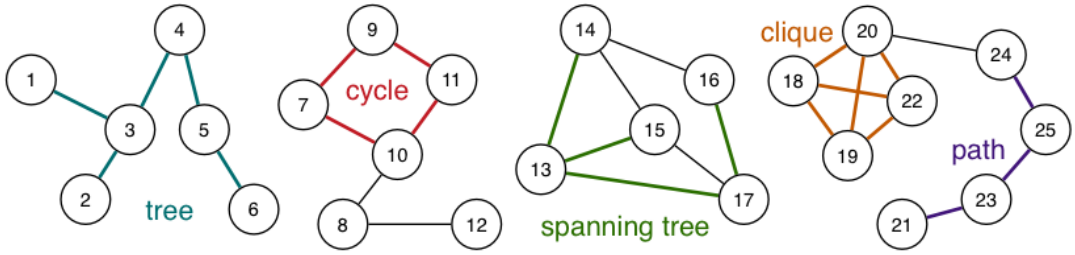


Cycle: 1–2, 2–3, 3–4, 4–1

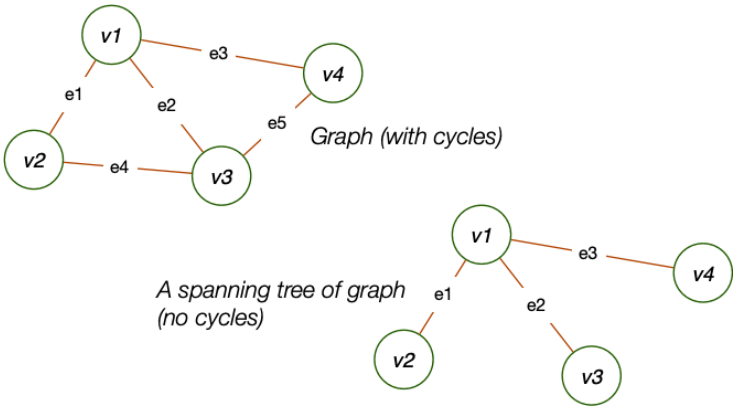
- Connected graph: There is a path from each vertex to every other vertex
- Complete graph: There is an edge from each vertex to every other vertex



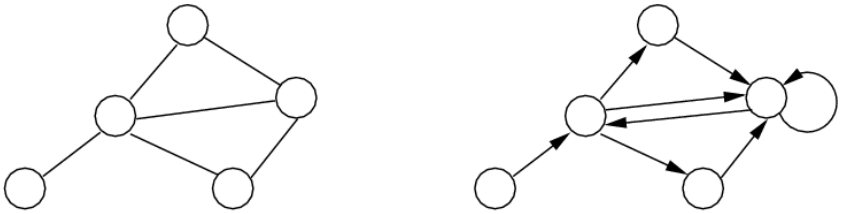
- Tree: Connected (sub)graph with no cycles
- Spanning tree: Tree containing all vertices
- Clique: Complete subgraph
- Consider the following as a single graph
    - This graph has 25 edges, 32 edges and 4 connected components



- A spanning tree: A subgraph of G containing all of V and is a single tree (connected, no cycles)
- A spanning forest: A subgraph of G containing all of V and is a set of trees (not connected, no cycles)



- Undirected graph: No self loops
- Directed graph: Can have self loops

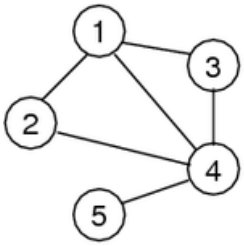


Undirected graph *Directed graph*

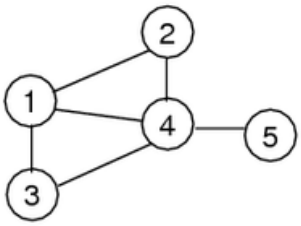
- Weighted graph: Each edge has an associated value (weight)
- Multi-graph: Allow multiple edges between two vertices
- Labelled graph: Edges have associated labels

5.1: Graph Representations

- Four representations of the same graph:



(a)



(b)

1–2 1–3 1–4
2–4
3–4
4–5

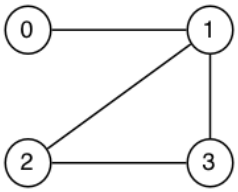
(c)

1–3
2–1 2–4
4–1 4–3
5–4

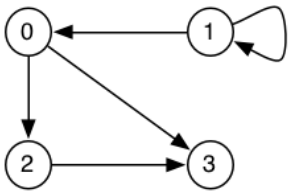
(d)

Array of Edges

- Edges are representations of Edge values (= pairs of verticles)
  - Space efficient
  - Adding and deleting edges is slightly complex
  - Undirected: Order of vertices in an Edge doesn't matter
  - Directed: Order of vertices in an Edge encodes direction



[(0,1), (1,2), (1,3), (2,3)]



[(1,0), (1,1), (0.2), (0,3), (2,3)]

- Assume edges is numbered 0 – (V – 1)
- Graph intialisation:

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph (no edges)
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate enough memory for g.edges[]
|   return g
```

- Assumes struct Graph { int nV; int nE; Edge edges[]; }
- Edge insertion:

```
insertEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g containing (v,w)
|
|   i=0
|   while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|      i=i+1
|   end while
|   if i=g.nE then       // (v,w) not found
|      g.edges[i]=(v,w)
|      g.nE=g.nE+1
|   end if
```

- Edge removal

```
removeEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g without (v,w)
|
|   i=0
|   while i < g.nE ∧ g.edges[i] ≠ (v,w) do
|      i=i+1
|   end while
|   if i < g.nE then  // (v,w) found
|      g.edges[i]=g.edges[g.nE-1]
|          // replace by last edge in array
|      g.nE=g.nE-1
|   end if
```

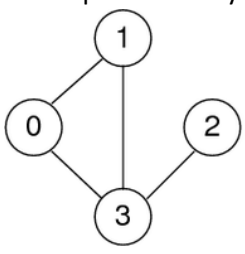
- Print a list of edges

```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nE-1 do
|   |   (v,w)=g.edges[i]
|   |   print v"—"w
|   end for
```

- Array-of-edges Cost Analysis
  - Storage cost: O(E)
  - Cost of operations:
    - Initialisation: O(1)
    - Insert edge: O(E)
    - Delete edge: O(E)
  - If array is full on insert
    - Allocate space for a bigger array, copy edges across -> O(E)
  - If we maintain edges in order
    - Use binary search to find edge -> O(log E)

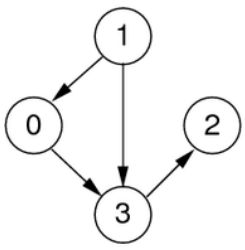
Adjacency Matrix Representation

- Represented by V x V matrix



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

- Advantages:
  - Easily implemented as 2-dimensional array
  - Can represent graphs, digraphs and weighted graphs
    - Graphs: Symmetric Boolean matrix
    - Digraphs: Non-symmetric Boolean matrix
    - Weighted: Non-symmetric matrix of weight values
- Disadvantages:
  - If few edges (sparse) -> memory inefficient ($O(V^2)$ space)
- Graph initialisation

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate memory for g.edges[][]
|   for all i,j=0..V-1 do
|      g.edges[i][j]=0    // false
|   end for
|   return g
```

- Edge insertion

```
insertEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g containing (v,w)
|
|   if g.edges[v][w] = 0 then  // (v,w) not in graph
|      g.edges[v][w]=1     // set to true
|      g.edges[w][v]=1
|      g.nE=g.nE+1
|   end if
```

- Edge removal

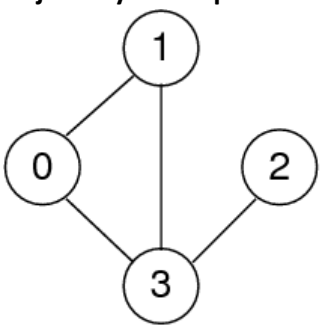
```
removeEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g without (v,w)
|
|   if g.edges[v][w] ≠ 0 then  // (v,w) in graph
|      g.edges[v][w]=0        // set to false
|      g.edges[w][v]=0
|      g.nE=g.nE-1
|   end if
```

- Print a list of edges

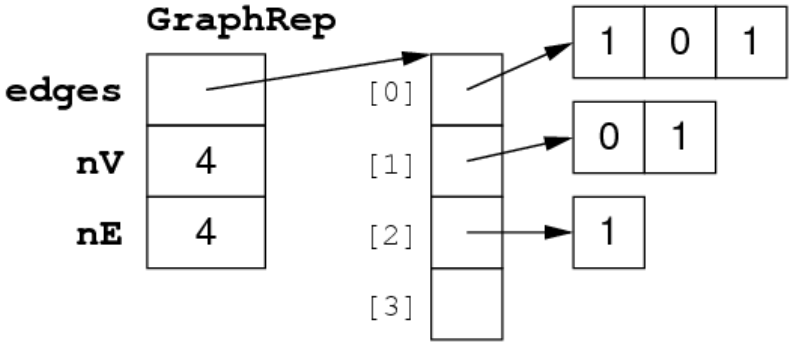
```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nV-1 do
|   |  for all j=i+1 to g.nV-1 do
|   |     if g.edges[i][j] ≠ 0 then
|   |        print i"—"j
|   |     end if
|   |  end for
|   end for
```

- Cost Analysis
    - Storage cost: $O(V^2)$
    - If the graph is sparse, most storage is wasted
    - Cost of operations:
        - Initialisation: $O(V^2)$
        - Insert edge: O(1)
        - Delete edge: O(1)

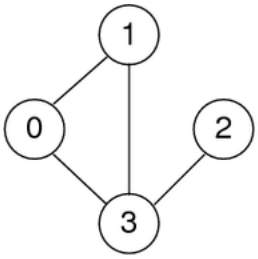
Adjacency List Representation



Undirected graph



- New storage cost: V-1 int ptrs + V(V+1)/2 ints
- Requires us to always use edges (v, w) such that v < w



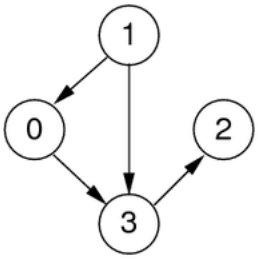
Undirected graph

A[0] = <1, 3>

A[1] = <0, 3>

A[2] = <3>

A[3] = <0, 1, 2>



Directed graph

A[0] = <3>

A[1] = <0, 3>

A[2] = < >

A[3] = <2>

- Advantages:
    - Relatively easy to implement in languages like C
    - Can represent graphs and digraphs
    - Memory efficient if E:V relatively small
- Disadvantages:
    - One graph has many possible representations

- Graph initialisation:

```
newGraph(V):
|   Input  number of nodes V
|   Output new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate memory for g.edges[]
|   for all i=0..V-1 do
|      g.edges[i]=newList()  // empty list
|   end for
|   return g
```

- Edge insertion:

```
insertEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g containing (v,w)
|
|   if not ListMember(g.edges[v],w) then
|      // (v,w) not in graph
|      ListInsert(g.edges[v],w)
|      ListInsert(g.edges[w],v)
|      g.nE=g.nE+1
|   end if
```

- Edge removal:

```
removeEdge(g,(v,w)):
|   Input  graph g, edge (v,w)
|   Output graph g without (v,w)
|
|   if ListMember(g.edges[v],w) then
|      // (v,w) in graph
|      ListDelete(g.edges[v],w)
|      ListDelete(g.edges[w],v)
|      g.nE=g.nE-1
|   end if
```

- Print a list of edges

```
showEdges(g):
|   Input graph g
|
|   for all i=0 to g.nV-1 do
|   |  for all v in g.edges[i] do
|   |     if i < v then
|   |        print i"—"v
|   |     end if
|   |  end for
|   end for
```

- Adjacency List Cost Analysis
  - Storage cost: O(V+E)
  - Cost of operations:
    - Initialisation: O(V)
    - Inserted edge: O(V)
    - Delete edge: O(V)

Comparison of Graph Representations

	array of edges	adjacency matrix	adjacency list
space usage	E	V^2	$V+E$
initialise	1	V^2	V
insert edge	E	1	E
remove edge	E	1	E

- Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	E	V	1
isPath(x,y)?	$E \cdot \log V$	V^2	$V+E$
copy graph	E	V^2	$V+E$
destroy graph	1	V	$V+E$

5.3: Graph ADT

- Data: set of edges, set of vertices
- Operations:
  - Building: create graph, add edge
  - Deleting: remove edge, drop whole graph
  - Scanning: Check if graph contains a given edge
- Things to note:
  - Set of vertices is fixed when graph initialised
  - We treat vertices as Ints but could be arbitrary Items

Graph.h

```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);  // new graph with V vertices
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex);
      // is there an edge between two vertices?
void  freeGraph(Graph);
```

Graph ADT (Array of Edges)

```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;    // #vertices (numbered 0..nV-1)
   int   nE;    // #edges
   int   n;     // size of edge array
} GraphRep;
```

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V; g->nE = 0;
   // allocate enough memory for edges
   g->n = Enough;
   g->edges = malloc(g->n*sizeof(Edge));
   assert(g->edges != NULL);
   return g;
}
```

```
// check if two edges are equal
bool eq(Edge e1, Edge e2) {
   return ( (e1.v == e2.v && e1.w == e2.w)
            || (e1.v == e2.w && e1.w == e2.v) );
```

```
}

// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

// check if an edge is valid in a graph
bool validE(Graph g, Edge e) {
   return (g != NULL && validV(e.v) && validV(e.w));
}
```

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists and array of edges isn't full
   assert(g != NULL && g->nE < g->n && isValidE(g,e));
   int i = 0;  // can't define in for (...)
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE)  // edge e not found
      g->edges[g->nE++] = e;
}
```

```
void removeEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i < g->nE)  // edge e found
      g->edges[i] = g->edges[--g->nE];
}
```

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));
   Edge e;
   e.v = x; e.w = y;
   for (int i = 0; i < g->nE; i++) {
      if (eq(e,g->edges[i]))  // edge found
         return true;
   }
   return false;  // edge not found
}
```

```
void insertEdge(Graph g, Edge e) {
   // ensure that g exists
   assert(g != NULL && validE(g,e));
   int i = 0;
   for (i = 0; i < g->nE; i++)
       if (eq(e,g->edges[i])) break;
   if (i == g->nE) { // edge e not found
      if (g->n == g->nE) {  // array full; expand
         g->edges = realloc(g->edges, 2*g->n);
         assert(g->edges != NULL);
         g->n = 2*g->n;
      }
      g->edges[g->nE++] = e;
   }
}
```

```
void freeGraph(Graph g) {
   assert(g != NULL);
   free(g->edges);  // free array of edges
   free(g);         // remove Graph object
}
```

Graph ADT (Adjacency Matrix)

```
typedef struct GraphRep {
   int  **edges; // adjacency matrix
   int    nV;    // #vertices
   int    nE;    // #edges
} GraphRep;
```

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate array of pointers to rows
   g->edges = malloc(V * sizeof(int *));
   assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (int i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int));
      assert(g->edges[i] != NULL);
   }
   return g;
}
```

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}
```

```
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));

   if (g->edges[e.v][e.w]) {   // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return (g->edges[x][y] != 0);
}
```

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      // free one row of matrix
      free(g->edges[i]);
   free(g->edges);  // free array of row pointers
   free(g);         // remove Graph object
}
```

Graph ADT (Adjacency Lists)

```
typedef struct GraphRep {
   Node **edges;  // array of lists
   int    nV;     // #vertices
   int    nE;     // #edges
} GraphRep;
```

```
typedef struct Node {
   Vertex v;
   struct Node *next;
} Node;
```

```
bool inLL(Node *L, Vertex v) {
   while (L != NULL) {
      if (L->v == v) return true;
      L = L->next;
   }
   return false;
}
```

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));
   assert(g != NULL);
   g->nV = V;  g->nE = 0;
   // allocate memory for array of lists
   g->edges = malloc(V * sizeof(Node *));
   assert(g->edges != NULL);
   for (int i = 0; i < V; i++)
```

```
      g->edges[i] = NULL;
   return g;
}
```

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (!inLL(g->edges[e.v], e.w)) {  // edge e not in graph
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      g->nE++;
   }
}
void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validE(g,e));
   if (inLL(g->edges[e.v], e.w)) {  // edge e in graph
      g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
      g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
      g->nE--;
   }
}
```

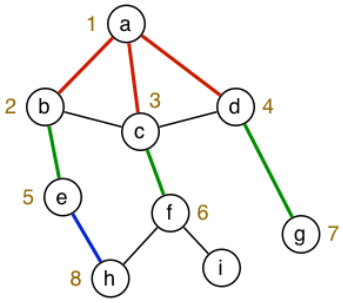
```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return inLL(g->edges[x], y);
}
```

```
void freeGraph(Graph g) {
   assert(g != NULL);
   for (int i = 0; i < g->nV; i++)
      freeLL(g->edges[i]); // free one list
   free(g->edges);  // free array of list pointers
   free(g);         // remove Graph object
}
```

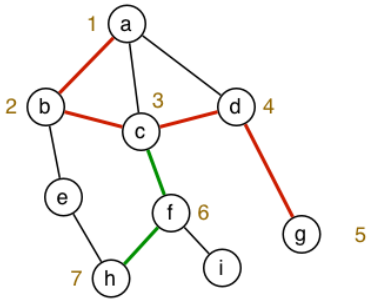
5.4: Graph Traversal

Graph Traversal

- Depth-first search (DFS)
  - Favours following path rather than neighbours
  - Implemented recursively or iteratively (via stack)
  - Full traversal produces a dept-first spanning tree
- Breadth-first search (BFS)
  - Favours neighbours rather than path following
  - Can be implemented iteratively (via queue)
  - Full traversal produces a breadth-first spanning tree



Breadth-first Search

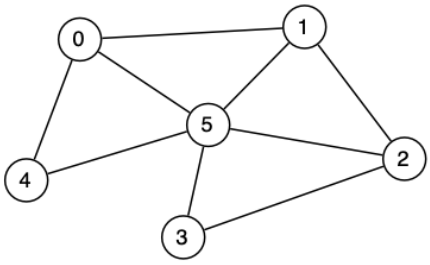


Depth-first Search

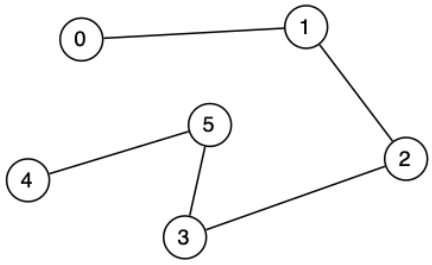
Spanning Tree

- A spanning tree of a graph includes all vertices, using a subset of edges, without cycles

Original graph

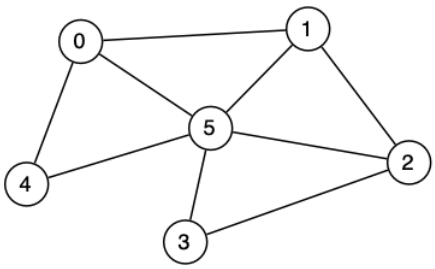


Spanning tree

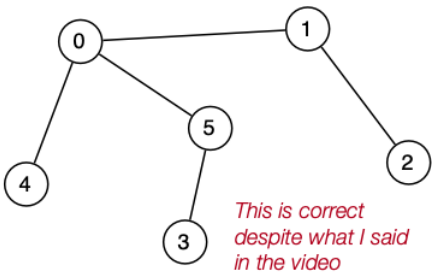


DFS Traversal: 0 -> 1 -> 2 -> 3 -> 5 -> 4

Original graph



Spanning tree



This is correct despite what I said in the video

BFS Traversal: 0 -> 1,4,5; 1 -> 2; 5 -> 3

Depth-first search

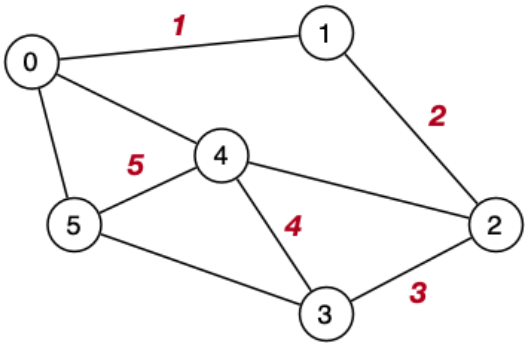
```
visited = {}

depthFirst(G,v):
|  visited = visited U {v}
|  for all (v,w) ∈ edges(G) do
|  |  if w ∉ visited then
|  |  |  depthFirst(G,w)
|  |  end if
|  end for
```

- Path checking:

```
visited = {}

hasPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if there is a path from src to dest,
|         false otherwise
|
|  return dfsPathCheck(G,src,dest)
```

```
dfsPathCheck(G,v,dest):
|  visited = visited U {v}
|  for all (v,w) ∈ edges(G) do
|  |  if w=dest then   // found edge to dest
|  |     return true
|  |  else if w ∉ visited then
|  |     if dfsPathCheck(G,w,dest) then
|  |        return true  // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false  // no path from v to dest
```

- Path check example: 0 -> 5



- Cost analysis:
  - Each vertex visited at most once, cost = O(V)
  - Visit all edges incident on visited vertices, cost = O(E)
    - Assuming an adjacency list representation
- Time complexity of DFS: O(V+E) (adjacency list representation)

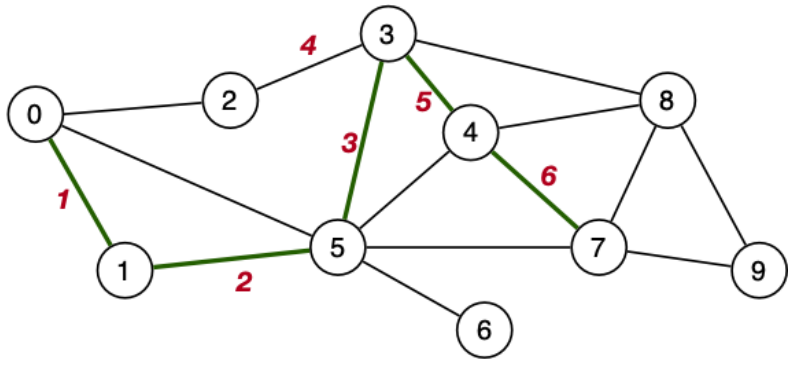
Path Finding

- Depth-first search can find a path to a node

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPath(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  visited[src]=src  // starting node of the path
|  if dfsPathCheck(G,src,dest) then
|  |  // show path in dest..src order
|  |  v=dest
|  |  while v≠src do
|  |     print v"-"
|  |     v=visited[v]
|  |  end while
|  |  print src
|  end if
```

```
dfsPathCheck(G,v,dest):
|  for all (v,w) ∈ edges(G) do
|  |  if visited[w] = -1 then
|  |     visited[w] = v
|  |     if w = dest then  // found edge from v to dest
|  |        return true
|  |     else if dfsPathCheck(G,w,dest) then
|  |        return true  // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false  // no path from v to dest
```

- The visited[] array after dfsPathCheck(G, 0, 7) succeeds



visited	0	0	3	5	3	1	5	4	-1	-1
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

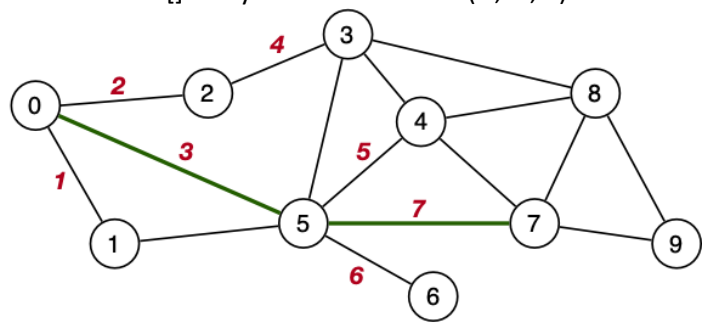
https://www.cse.unsw.edu.au/~cs2521/19T2/lecs/week04b/dfs_bfs_trace/Graph_DFS.htm

Breadth-first Search

- Visit and mark current vertex
- Visit all neighbours of current vertex
- Then consider neighbours of neighbours
- Uses queue

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPathBFS(G,src,dest):
|  Input  graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  enqueue src into queue Q
|  while not found ∧ Q is not empty do
|  |  dequeue v from Q
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w) ∈ edges(G) with visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     enqueue w into Q
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

- The visited[] array after findPathBFS(G, 0 , 7) succeeds



visited	0	0	0	2	5	0	5	5	-1	-1
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

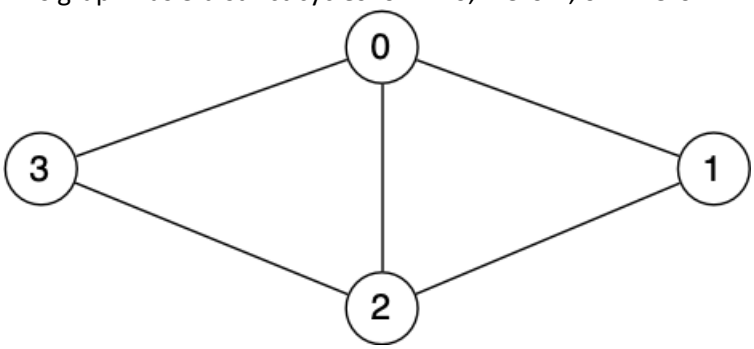
https://www.cse.unsw.edu.au/~cs2521/19T2/lecs/week04b/dfs_bfs_trace/Graph_BFS.htm

- Cost Analysis:
  - O(V+E) same as DFS

5.5: Graph Algorithms

Cycle Checking

- A graph has a cycle if:
  - It has a path of length > 2
  - With start vertex *src* = end vertex *dest*
  - And without using any edge more than once
- This graph has 3 distinct cycles: 0-1-2-0, 2-3-0-2, 0-1-2-3-0




```
hasCycle(G):
|  Input  graph G
|  Output true if G has a cycle, false otherwise
|
|  choose any vertex v ∈ G
|  return dfsCycleCheck(G,v)

dfsCycleCheck(G,v):
|  mark v as visited
|  for each (v,w) ∈ edges(G) do
|  |  if w has been visited then  // found cycle
|  |     return true
|  |  else if dfsCycleCheck(G,w) then
|  |     return true
```

```
bool dfsCycleCheck(Graph g, Vertex v, Vertex u) {
   visited[v] = true;
   for (Vertex w = 0; w < g->nV; w++) {
      if (adjacent(g, v, w)) {
         if (!visited[w]) {
            if (dfsCycleCheck(g, w, v))
               return true;
         }
         else if (w != u)
            return true;
      }
   }
   return false;
}
```

```
Vertex *visited;

bool hasCycle(Graph g, Vertex s) {
   bool result = false;
   visited = calloc(g->nV,sizeof(int));
```

```
   for (int v = 0; v < g->nV; v++) {
      for (int i = 0; i < g->nV; i++)
         visited[i] = -1;
      if dfsCycleCheck(g, v, v)) {
         result = true;
         break;
      }
   }
   free(visited);
   return result;
}
```

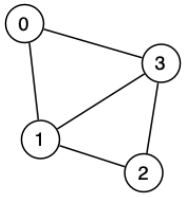
- But what about non-connected components of a graph?
    - Add new fields to the GraphRep structure

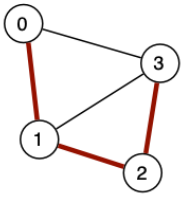
```
typedef struct GraphRep *Graph;

struct GraphRep {
   ...
   int nC;  // # connected components
   int *cc; // which component each vertex is contained in
   ...      // i.e. array [0..nV-1] of 0..nC-1
}
```

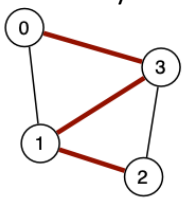
```
// How many connected subgraphs are there?
int nConnected(Graph g) {
   return g->nC;
}
// Are two vertices in the same connected subgraph?
bool inSameComponent(Graph g, Vertex v, Vertex w) {
   return (g->cc[v] == g->cc[w]);
}
```

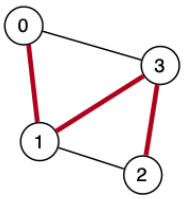
Hamiltonian Path and Circuit

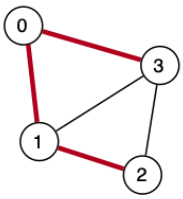
- Find a simple path connected two vertices v, w in a graph G
- Such that the path includes every vertex exactly once

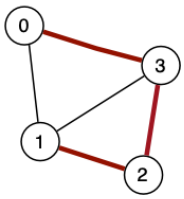












- Approach:
    - Generate all possible simple paths (using e.g. DFS)
    - Keep a counter of vertices visited in current path
    - Stop when find a path containing V vertices
- Can be expressed via a recursive DFS algorithm

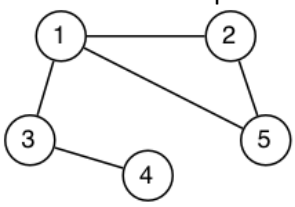
```
visited[] // array [0..nV-1] to keep track of visited vertices

hasHamiltonianPath(G,src,dest):
|  Input  graph G, plus src/dest vertices
|  Output true if Hamiltonian path src...dest,
|           false otherwise
|
|  for all vertices v ∈ G do
|     visited[v]=false
|  end for
|  return false  // no cycle at v
```

```
|  return hamiltonR(G,src,dest,#vertices(G)-1)
hamiltonR(G,v,dest,d):
|  Input G    graph
|        v    current vertex considered
|        dest destination vertex
|        d    distance "remaining" until path found
|
|  if v=dest then
|     if d=0 then return true else return false
|  else
|  |  visited[v]=true
|  |  for each (v,w) ∈ edges(G)  where not visited[w] do
|  |     if hamiltonR(G,w,dest,d-1) then
|  |        return true
|  |     end if
|  |  end for
|  end if
|  visited[v]=false           // reset visited mark
|  return false
```

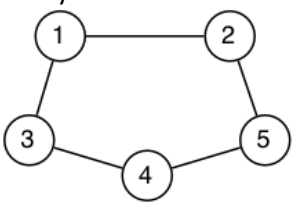
- Analysis: Worst case requires (V-1)! paths to be examined

Euler Path and Circuit

- Find a path connected two vertices v, w in graph G
- Such that the path includes each edge exactly once



Euler Path: 4-3-1-5-2-1



Euler Circuit: 1-2-5-4-3-1

```
hasEulerPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if G has Euler path from src to dest
|         false otherwise
|
|  if src≠dest then
|     if degree(G,src) is even ∨ degree(G,dest) is even then
|        return false
|     end if
|  end if
|  for all vertices v ∈ G do
|     if v≠src ∧ v≠dest ∧ degree(G,v) is odd then
|        return false
|     end if
|  end for
|  return true
```

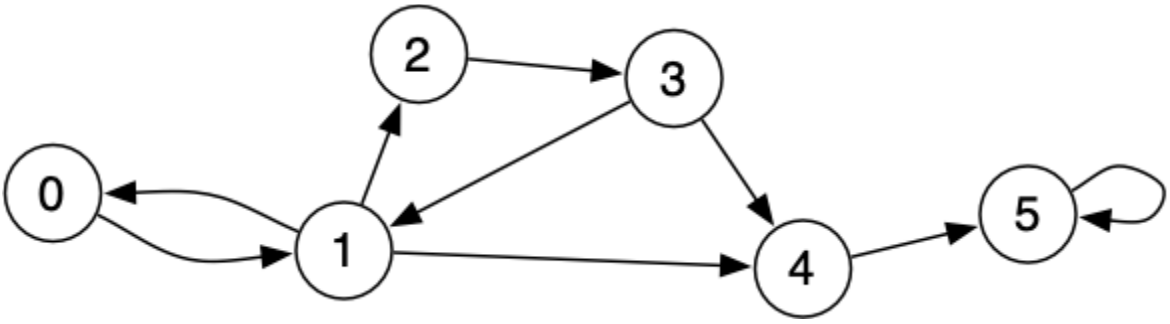
- Overall cost is $O(V^2)$

Week 7

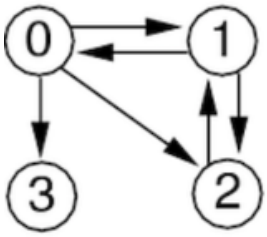
7.1: Directed/Weighted Graphs

Directed Graphs (Digraphs)

- Example:



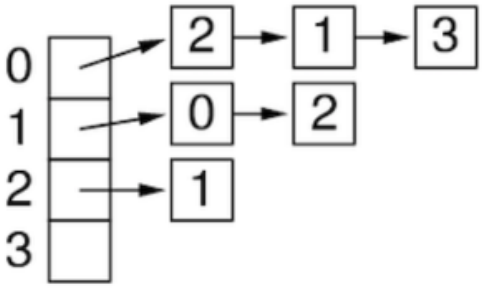
- Some properties of:
  - Edges 1-2-3 form a cycle, edges 1-3-4 do not form a cycle
  - Vertex 5 has a self-referencing edge (5, 5)
  - Vertices 0 and 1 reference each other, i.e. (0, 1) and (1, 0)
  - There are no paths from 5 to any other nodes
  - Paths from -> 5: 0-1-2-3-4-5, 0-1-4-5, 0-1-2-3-1-4-5
- Outdegree: Number of edges coming out of a node
- Indegree: Number of edges coming into the node



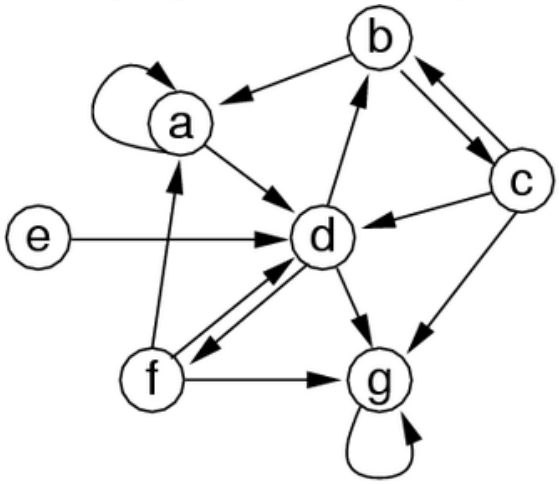
digraph

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	0	1	0	0
3	0	0	0	0

adj matrix



adj lists



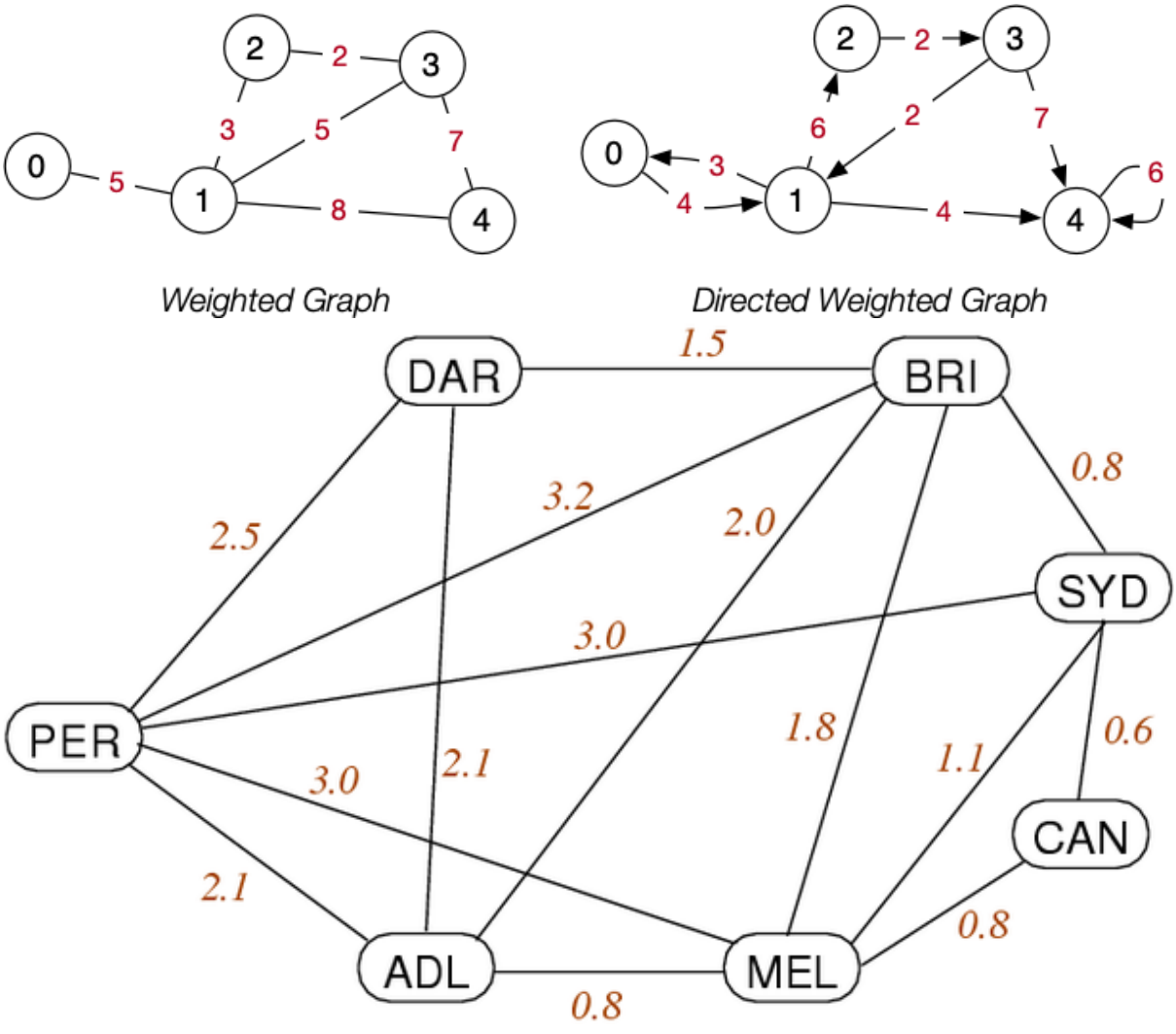
	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

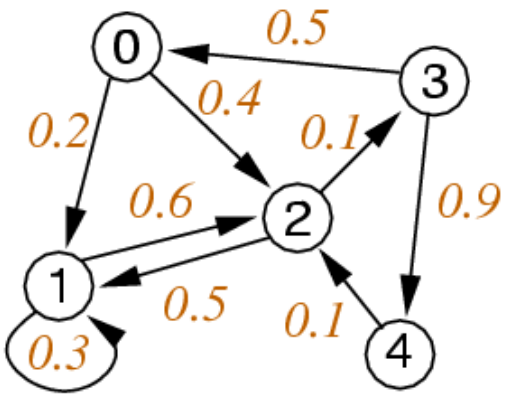
Costs of representations: (where degree *deg(v)* = #edges leaving *v*)

	array of edges	adjacency matrix	adjacency list
space usage	*E*	V^2	*V+E*
insert edge	*E*	*1*	*1*
exists edge *(v,w)*?	*E*	*1*	*deg(v)*
get edges leaving *v*	*E*	*V*	*deg(v)*

Overall, adjacency list representation is best

Weighted Graphs

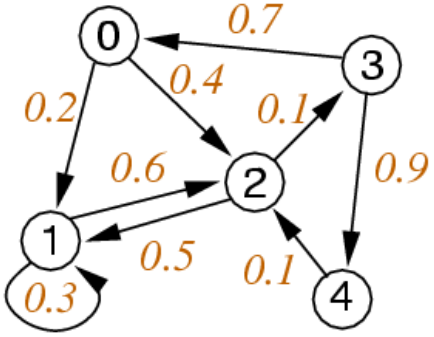




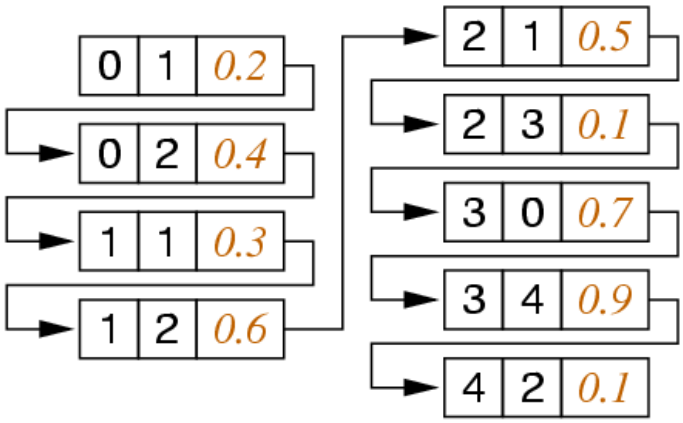
Weighted Digraph

	0	1	2	3	4
0	*	0.2	0.4	*	*
1	*	0.3	0.6	*	*
2	*	0.5	*	0.1	*
3	0.5	*	*	*	0.9
4	*	*	0.1	*	*

Adjacency Matrix



Weighted Digraph



Edge List

```
// edges are pairs of vertices (end-points) plus weight
typedef struct Edge {
   Vertex v;
   Vertex w;
   int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

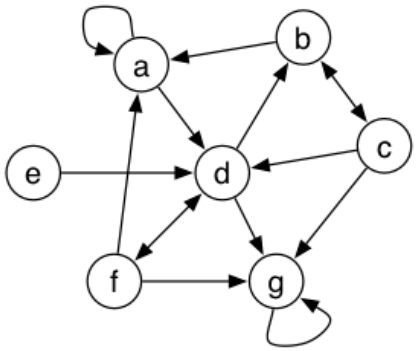
```
typedef struct GraphRep {
   int **edges;  // adjacency matrix storing weights
                 // 0 if nodes not adjacent
   int   nV;     // #vertices
   int   nE;     // #edges
} GraphRep;

bool adjacent(Graph g, Vertex v, Vertex w) {
   assert(valid graph, valid vertices)
   return (g->edges[v][w] != 0);
}
```

```
void insertEdge(Graph g, Edge e) {
   assert(valid graph, valid edge)
   // edge e not already in graph
   if (g->edges[e.v][e.w] == 0) g->nE++;
   // may change weight of existing edge
   g->edges[e.v][e.w] = e.weight;
   g->edges[e.w][e.v] = e.weight;
}

void removeEdge(Graph g, Edge e) {
   assert(valid graph, valid edge)
   // edge e not in graph
   if (g->edges[e.v][e.w] == 0) return;
   g->edges[e.v][e.w] = 0;
   g->edges[e.w][e.v] = 0;
   g->nE--;
}
```

7.2: Digraph Algorithms

Transitive Closure

- Goal: Determine reachability, is there a path from t to s?
- Transitive closure:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

adjacency matrix

	a	b	c	d	e	f	g
a	1	1	1	1	0	1	1
b	1	1	1	1	0	1	1
c	1	1	1	1	0	1	1
d	1	1	1	1	0	1	1
e	1	1	1	1	0	1	1
f	1	1	1	1	0	1	1
g	0	0	0	0	0	0	1

reachability matrix

- Warshall's Algorithm:

```
makeTC(G):
|  tc[][] = edges[][]
|  for all i ∈ vertices(G) do
|  |  for all s ∈ vertices(G) do
|  |  |  for all t ∈ vertices(G) do
|  |  |  |  if tc[s][i]=1 ∧ tc[i][t]=1 then
|  |  |  |  |  tc[s][t]=1
|  |  |  |  end if
|  |  |  end for
|  |  end for
|  end for
```

- Cost: $O(V^3)$

```
depthFirst(G,v):
|  mark v as visited
|  for each (v,w) ∈ edges(G) do
|  |  if w has not been visited then
|  |  |  depthFirst(w)
|  |  end if
|  end for

breadthFirst(G,v):
|  enqueue v
|  while queue not empty do
|  |  curr=dequeue
|  |  if curr not already visited then
|  |  |  mark curr as visited
|  |  |  enqueue each w where (curr,w) ∈ edges(G)
|  |  end if
```

Page Rank

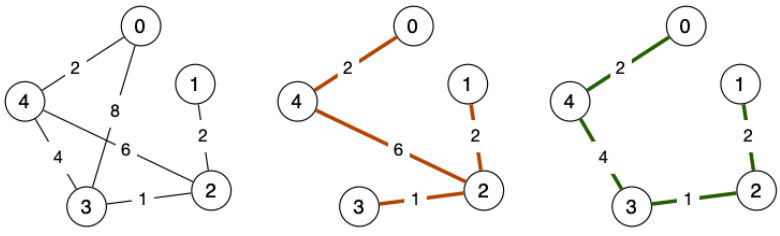
- Goal: To determine which Web pages are 'important'
- Approach: Ignore page contents; focus on hyperlinks
    - Treat Web as a graph: page = vertex, hyperlink = edge

```
PageRank(myPage):
|   rank=0
|   for each page in the Web do
|   |   if linkExists(page,myPage) then
|   |      rank=rank+1
|   |   end if
|   end for
```

- Random web surfer strategy:
    - Each age typically has many outbound hyperlinks …
        - Choose one at random, without a visited[] check
        - Follow link and repeat above process on destination page
    - If no visited check, need a way to (mostly) avoid loops
- Important property of this strategy:
    - If we randomly follow links in the webpage
    - … more likely to re-discover pages with many inbound links

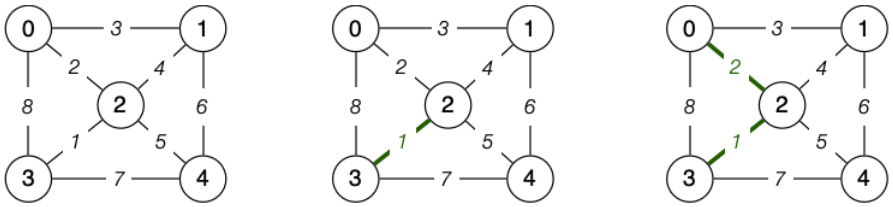
```
curr=random page, prev=null
for a long time do
|   if curr not in array rank[] then
|      rank[curr]=0
|   end if
|   rank[curr]=rank[curr]+1
|   if random(0,100) < 85 then // with 85% chance ...
|      prev=curr               // ... keep crawling
|      curr=choose hyperlink from curr
|   else
|      curr=random page, not prev // avoid getting stuck
|      prev=null
|   end if
end for
```

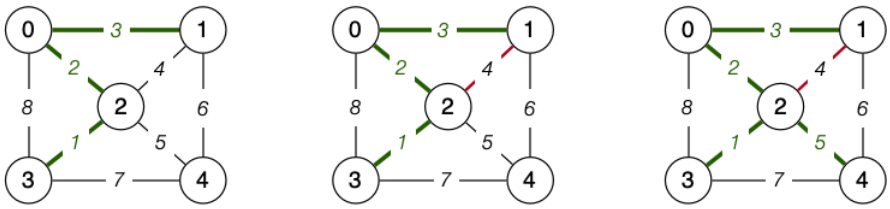
7.3: Minimum Spanning Tree



Kruskal's Algorithm

- Start with empty MST
- Consider edges in increasing weight order
    - Add edge if it does not form a cycle in MST
- Repeat until V-1 edges are added
- Critical operations:
    - Iterating over edges in weight order
    - Checking for cycles in a graph



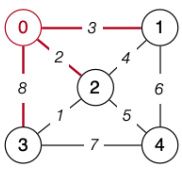



```
KruskalMST(G):
|   Input  graph G with n nodes
|   Output a minimum spanning tree of G
|
|   MST=empty graph
|   sort edges(G) by weight
|   for each e ∈ sortedEdgeList do
|   |   MST = MST ∪ {e}   // add edge
|   |   if MST has a cyle then
|   |      MST = MST \ {e}   // drop edge
|   |   end if
|   |   if MST has n-1 edges then
|   |      return MST
|   |   end if
|   end for
```

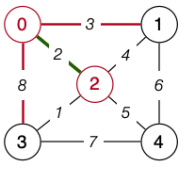
- Complexity analysis:
    - Sort edge list is O(E x log E)
    - At least V iterations over sorted edges
    - On each iteration:
        - Getting next lowest cost edge is O(1)
        - Checking whether adding it forms a cycle: cost = $O(V^2)$

Prim's Algorithm

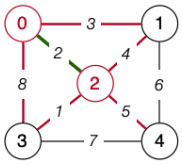
- Start from any vertex V and empty MST
- Choose edge not already in MST to add to MST; must be:
    - Node must not already be connected to MST
    - Minimal weight of all such edges
- Repeat until MST covers all vertices



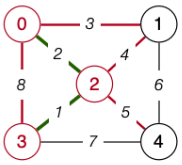
Start of step 1



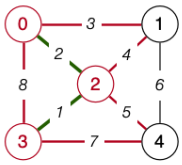
End of step 1



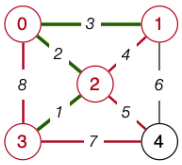
Start of step 2



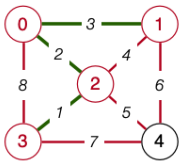
End of step 2



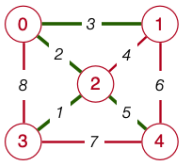
Start of step 3



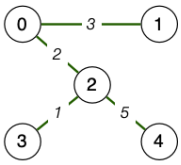
End of step 3



Start of step 4



End of step 4



MST

```
PrimMST(G):
|   Input  graph G with n nodes
|   Output a minimum spanning tree of G
|
|   MST=empty graph
|   usedV={0}
|   unusedE=edges(g)
|   while |usedV| < n do
|   |   find e=(s,t,w) ∈ unusedE such that {
|   |      s ∈ usedV ∧ t ∉ usedV
|   |        ∧ w is min weight of all such edges
|   |   }
|   |   MST = MST ∪ {e}
|   |   usedV = usedV ∪ {t}
|   |   unusedE = unusedE \ {e}
|   end while
|   return MST
```

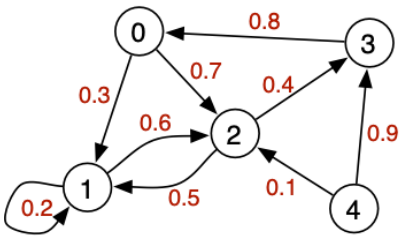
- Complexity analysis:

- V iterations of outer loop
- In each iteration, finding min-weighted edge…
    - With set of edges is O(E) -> O(V x E) overall
    - With priority queue is O(log E) -> O (V x log E) overall

7.4: Shortest Path

Single-source Shortest Path (SSSP)

- Shortest paths from s to all other vertices
- dist[] V-indexed array of cost of shortest path from s
- pred[] V- indexed array of predecessor in shortest path from s



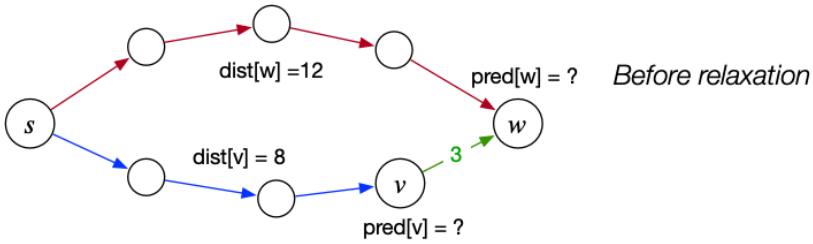
	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

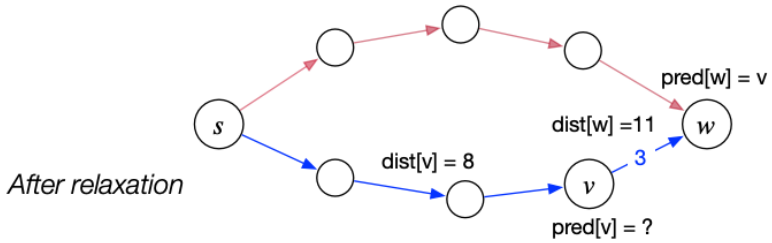
Shortest paths from s=0

	0	1	2	3	4
dist	1.3	0.6	0.1	0.5	0
pred	3	2	4	2	-

Shortest paths from s=4

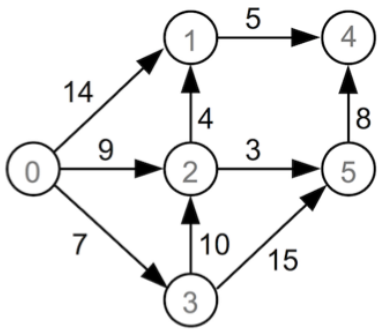
Edge Relaxation





Dijkstra's Algorithm

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s
vSet    // vertices whose shortest path from s is unknown

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise all dist[] to ∞
|  dist[source]=0
|  initialise all pred[] to -1
|  vSet=all vertices of G
|  while vSet ≠ ∅ do
|  |  find v ∈ vSet with minimum dist[v]
|  |  for each (v,w,weight) ∈ edges(G) do
|  |     relax along (v,w,weight)
|  |  end for
|  |  vSet=vSet \ {v}
|  end while
```




```
while vSet not empty do
  find v in vSet
    with min dist[v]
  for each (v,w,weight) in E do
     relax along (v,w,weight)
  end for
  vSet = vSet \ {v}
end while
```

Initially

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	inf	inf	inf	inf	inf
pred	-	-	-	-	-	-

First iteration, v=0

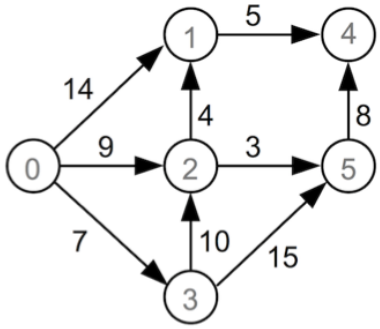
	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	int
pred	-	0	0	0	-	-

Second Iteration, v=3

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	14	9	7	inf	22
pred	-	0	0	0	-	3

Third iteration, v=2

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	inf	12
pred	-	2	0	0	-	2



Fourth iteration, v=5

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	20	12
pred	-	2	0	0	5	2

```
while vSet not empty do
  find v in vSet
    with min dist[v]
  for each (v,w,weight) in E do
     relax along (v,w,weight)
  end for
  vSet = vSet \ {v}
end while
```

Fifth iteration, v=1

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Sixth iteration, v=4

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	13	9	7	18	12
pred	-	2	0	0	1	2

Completed, vSet is empty

- Time complexity analysis:
  - Each edge needs to be considered once -> O(E)
  - Outer loop has O(V) iterations

Week 8

8.1: Sorting

Concrete Framework

```
// we deal with generic Items
typedef SomeType Item;

// abstractions to hide details of Items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B = t;}

// Sorts a slice of an array of Items, a[lo..hi]
void sort(Item a[], int lo, int hi);

// Check for sortedness (to validate functions)
int isSorted(Item a[], int lo, int hi);
```

```
bool isSorted(Item a[], int lo, int hi)
{
   for (int i = lo; i < hi; i++) {
      if (!less(a[i],a[i+1])) return false;
   }
   return true;
}
```

Sorts on Linux

- The sort command:
    - Sorts a file of text, understands fields in a line
    - Can sort alphabetically, numerically, reverse, random
- The qsort command:
    - qsort(void *a, int n, int size, int (*cmp)())
    - Sorts any kind of array
    - Requires the user to supply a comparison function
    - Sorts list of items using the order given by cmp()

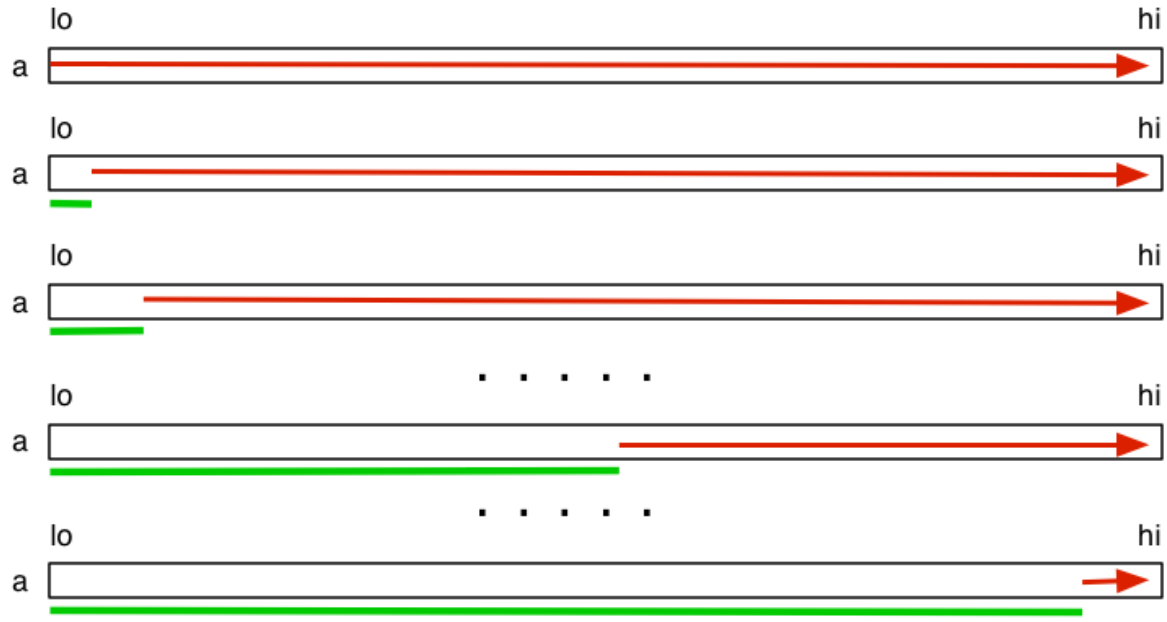
8.2: O(n^2) Sorts

O(n^2) Sorting Algorithms

- Selection sort: Simple, non-adaptive sort
- Bubble sort: Simple, adaptive sort
- Insertion sort: Simple, adaptive sort
- Shell sort: Improved version of insertion sort
- For small arrays, the above methods are adequate
- Adaptive: Time complexity depends on how 'sorted' the original list is

Selection Sort

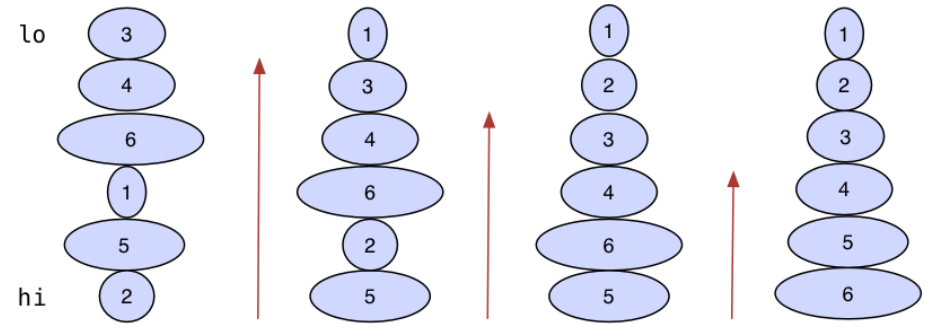
- Simple, non-adaptive method:
    - Find the smallest element, put it into first array slot
    - Find second smallest element, put it into second array sort
    - Repeat until all elements in the correct position

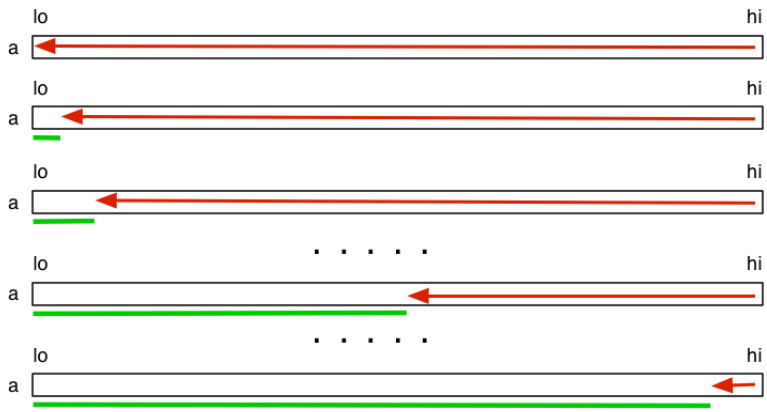



```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

Bubble Sort

- Simple adaptive method:
  - Make multiple passes from the highest to lowest
  - On each pair passed, swap if they are out of order



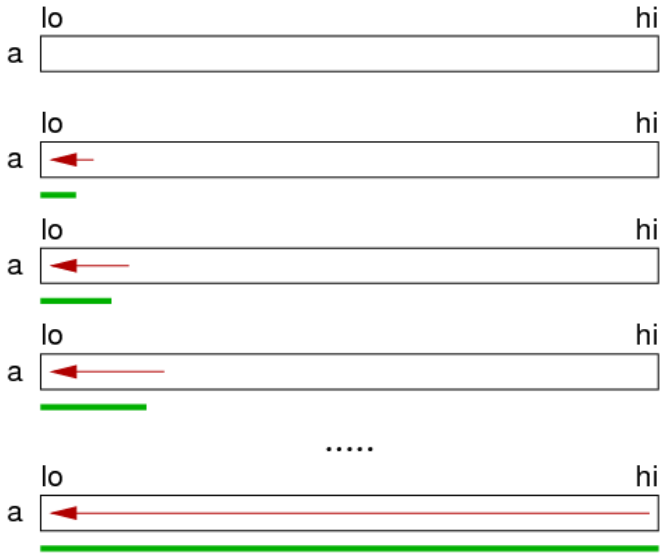



```
S O R T E X A M P L E
A S O R T E X E M P L
A E S O R T E X L M P
A E E S O R T L X M P
A E E L S O R T M X P
A E E L M S O R T P X
A E E L M O S P R T X
A E E L M O P S R T X
A E E L M O P R S T X
... no swaps ⇒ done ...
A E E L M O P R S T X
```

```
void bubbleSort(int a[], int lo, int hi)
{
   int i, j, nswaps;
   for (i = lo; i < hi; i++) {
      nswaps = 0;
      for (j = hi; j > i; j--) {
         if (less(a[j], a[j-1])) {
            swap(a[j], a[j-1]);
            nswaps++;
         }
      }
      if (nswaps == 0) break;
   }
}
```

Insertion Sort

- Simple adaptive method:
  - Take first element and treat as sorted array
  - Take next element and insert into sorted part of array so that order is preserved
  - Above increases length of sorted path by one
  - Repeat until whole array is sorted

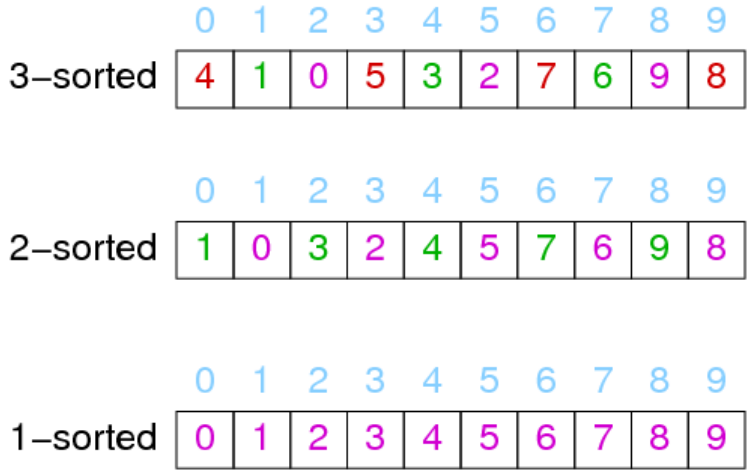



```
S O R T E X A M P L E
S O R T E X A M P L E
O S R T E X A M P L E
O R S T E X A M P L E
O R S T E X A M P L E
E O R S T X A M P L E
E O R S T X A M P L E
A E O R S T X M P L E
A E M O R S T X P L E
A E M O P R S T X L E
A E L M O P R S T X E
A E E L M O P R S T X
```

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (!less(val,a[j-1])) break;
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

ShellSort: Improving Insertion Sort

- Array is h-sorted if taking every h'th eelement yields a sorted array
- A h-sorted array is made up of n/h interleaved sorted arrays
- Shell sort: h-sort array for progressively smaller h, ending with 1-sorted




```
void shellSort(int a[], int lo, int hi)
{
   int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};
   int g, h, start, i, j, val;
   for (g = 0; g < 8; g++) {
      h = hvals[g];
      start = lo + h;
      for (i = start+1; i <= hi; i++) {
         val = a[i];
         for (j = i; j >= start; j -= h) {
            if (!less(val,a[j-h]) break;
            a[j] = a[j-h];
         }
         a[j] = val;
      }
   }
}
```

- Complexity may be $O(n^{1.5})$ or $O(n^{4/3})$

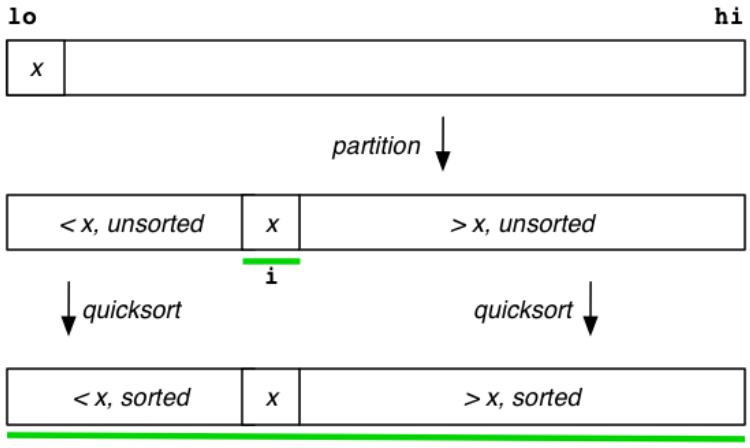
Sorting Comparison

	#compares			#swaps			#moves		
	min	avg	max	min	avg	max	min	avg	max
Selection sort	n^2	n^2	n^2	n	n	n	.	.	.
Bubble sort	n	n^2	n^2	0	n^2	n^2	.	.	.
Insertion sort	n	n^2	n^2	.	.	.	n	n^2	n^2
Shell sort	n	$n^{4/3}$	$n^{4/3}$	.	.	.	1	$n^{4/3}$	$n^{4/3}$

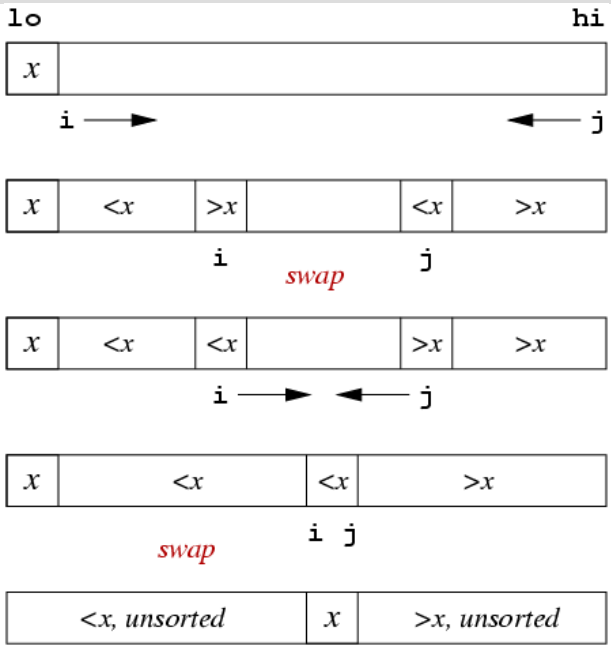
8.3: Quicksort

Quicksort

- Choose an item to be a pivot
- Rearrange (partition) the array so that:
    - All elements to left of pivot are smaller than pivot
    - All elements of pivot are greater than pivot
- Recursively sort each of the partitions




```
void quicksort(Item a[], int lo, int hi)
{
   int i; // index of pivot
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```



```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];  // pivot
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

Quicksort Performance

- Best case: O(n logn)
- Worst case: $O(n^2)$

Quicksort Improvements

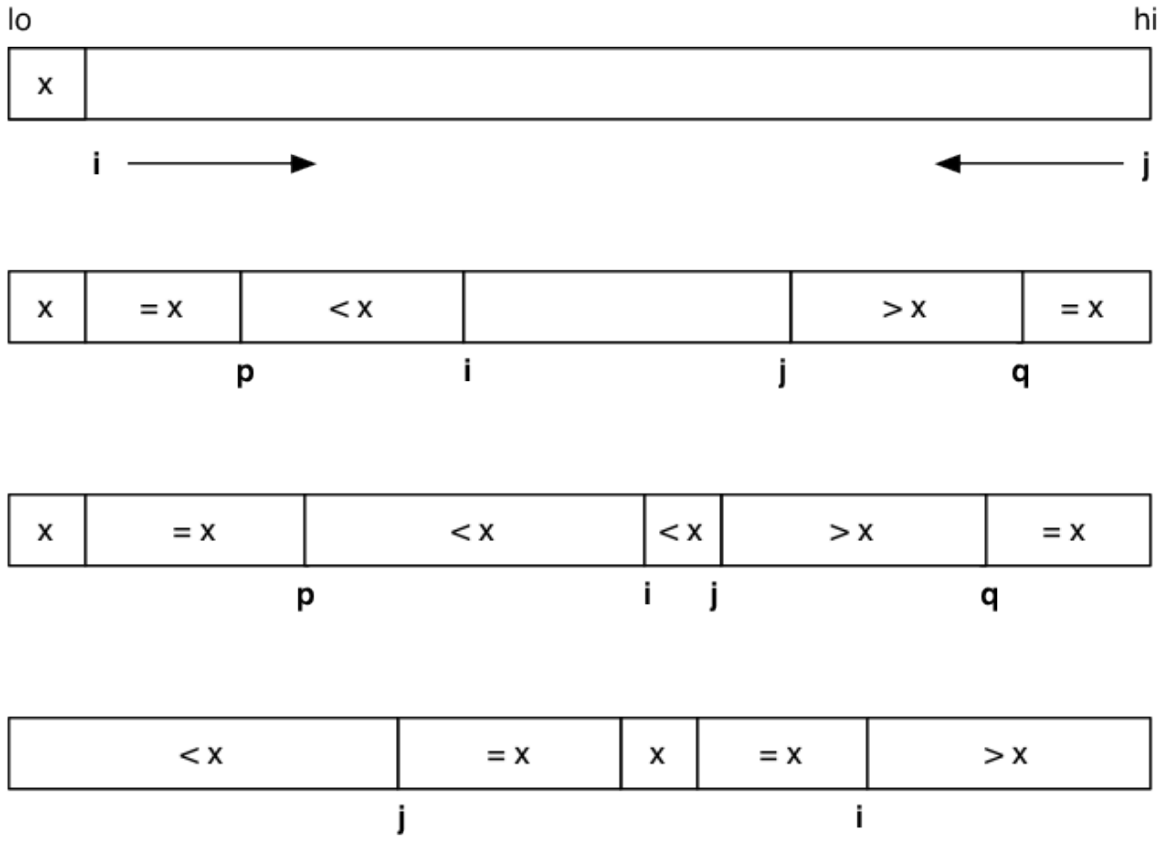
- Choice of pivot can have significant effect
    - Always choosing largest/smallest -> worst case
    - Find intermediate value by median of tree

```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   // now, we have a[lo] < a[mid] < a[hi]
   // swap a[mid] to a[lo+1] to use as pivot
   swap(a, mid, lo+1);
}
void quicksort(Item a[], int lo, int hi)
{
   if (hi <= lo) return;
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

- Little benefit when partitioning when size < 5
- So use another sort, e.g. insertionSort

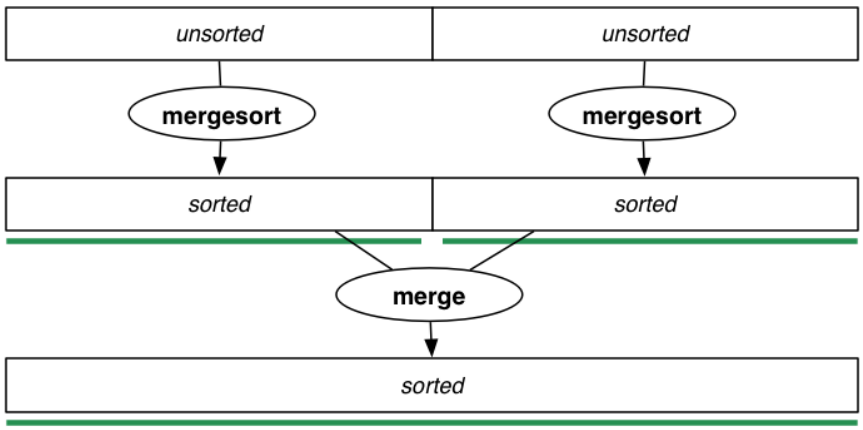
```
void quicksort(Item a[], int lo, int hi)
{
   if (hi-lo < Threshhold) {
      insertionSort(a, lo, hi);
      return;
   }
   medianOfThree(a, lo, hi);
   int i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Three Way Partitioning

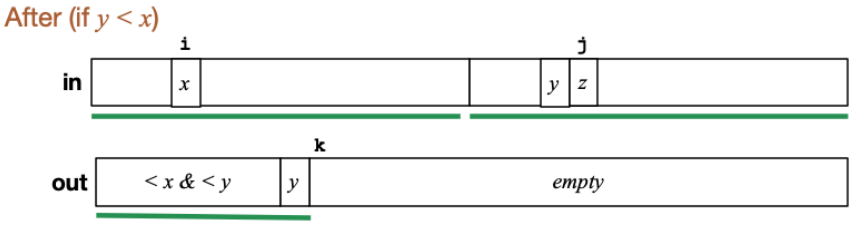


8.4: Mergesort

- Split the array into two equal-sized partitions
- Recursively sort each of the partitions
- Merge the two partitions into a new sorted array
- Copy back into original array




```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2; // mid point
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```



```
void merge(Item a[], int lo, int mid, int hi)
{
   int  i, j, k, nitems = hi-lo+1;
   Item *tmp = malloc(nitems*sizeof(Item));

   i = lo; j = mid+1; k = 0;
   // scan both segments, copying to tmp
   while (i <= mid && j <= hi) {
     if (less(a[i],a[j]))
        tmp[k++] = a[i++];
     else
        tmp[k++] = a[j++];
   }
   // copy items from unfinished segment
   while (i <= mid) tmp[k++] = a[i++];
   while (j <= hi) tmp[k++] = a[j++];

   //copy tmp back to main array
   for (i = lo, k = 0; i <= hi; i++, k++)
      a[i] = tmp[k];
   free(tmp);
}
```

- Time complexity O(N log N)

Bottom-up Mergesort

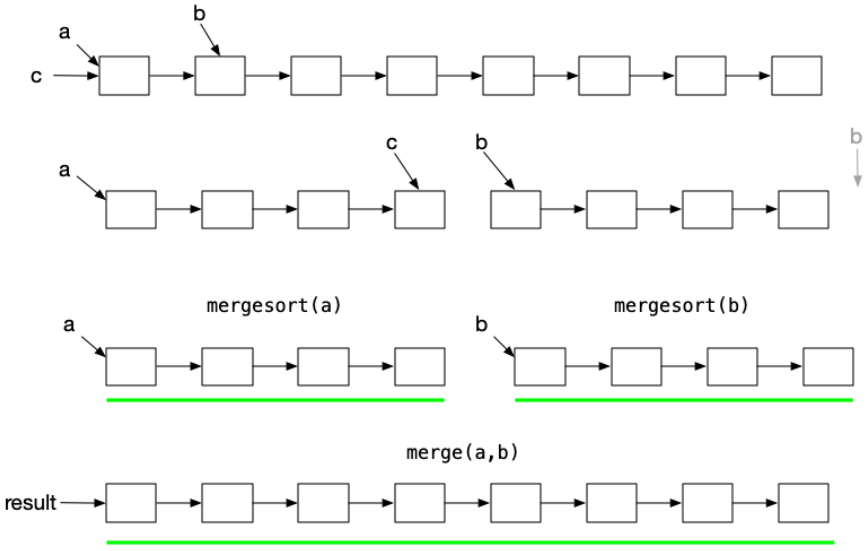
- Non-recursive merge sort does not require a stack
  - Partition boundaries can be computed iteratively
- Bottom-up mergesort:
  - On each pass, array contains sorted runs of length m
  - At start, treat as N sorted runs of length 1
  - 1st pass merges adjacent elements into runs of length 2
  - 2nd pass merges adjacent 2-runs into runs of length 4
  - Continue until a single sorted run of length N

```
#define min(A,B) ((A < B) ? A : B)

void mergesort(Item a[], int lo, int hi)
{
   int m;     // m = length of runs
   int len;   // end of 2nd run
   Item c[]; // array to merge into
   for (m = 1; m <= lo-hi; m = 2*m) {
      for (int i = lo; i <= hi-m; i += 2*m) {
         len = min(m, hi-(i+m)+1);
         merge(&a[i], m, &a[i+m], len, c);
         copy(&a[i], c, m+len);
      }
   }
}
```

```
// merge arrays a[] and b[] into c[]
// aN = size of a[], bN = size of b[]
void merge(Item a[], int aN, Item b[], int bN, Item c[])
{
   int i; // index into a[]
   int j; // index into b[]
   int k; // index into c[]
   for (i = j = k = 0; k < aN+bN; k++) {
      if (i == aN)
         c[k] = b[j++];
      else if (j == bN)
         c[k] = a[i++];
      else if (less(a[i],b[j]))
         c[k] = a[i++];
      else
         c[k] = b[j++];
   }
}
```

Mergesort and Linked Lists

```
List merge(List a, List b)
{
   List new = newList();
   while (a != NULL && b != NULL) {
      if (less(a->item, b->item))
         { new = ListAppend(new, a->item); a = a->next; }
      else
         { new = ListAppend(new, b->item); b = b->next; }
   }
   while (a != NULL)
      { new = ListAppend(new, a->item); a = a->next; }
   while (b != NULL)
      { new = ListAppend(new, b->item); b = b->next; }
   return new;
}
```



```
List mergesort(List c)
{
   List a, b;
   if (c == NULL || c->next == NULL) return c;
   a = c;  b = c->next;
   while (b != NULL && b->next != NULL)
      { c = c->next; b = b->next->next; }
   b = c->next; c->next = NULL;  // split list
   return merge(mergesort(a), mergesort(b));
}
```