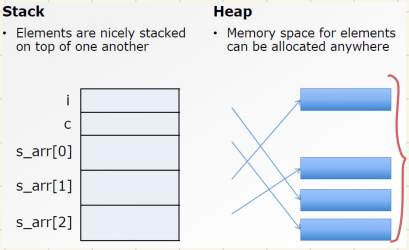
Dynamic Data Structures

★ Proff goes through structs in C



Dynamic Data, non-sequential

Malloc ( )

Dynamically allocate memory

```
typedef struct node stud_t{
        int age;
        float height;
};
stud_t *arr = (stud_t *) malloc (sizeof(stud_t)*6);
```

typecast the void pointer malloc returns into the stud_t type of arr

```
int *array = (int *) malloc(sizeof(int)*5);
```

```
free (arr);
```

Clear the memory after it has been used, however, the space for the pointer is still allocated to [arr]

- malloc() does not "clear" the data in the memory.
- If you would like to initialize the elements as zero,
  1. Write a loop to initialize them to zero
  2. Use calloc()

  Eg. int *array = (int *) calloc(5,sizeof(int));
- free() is still required to deallocate memory
- realloc() allows user change the size of the allocated memory.

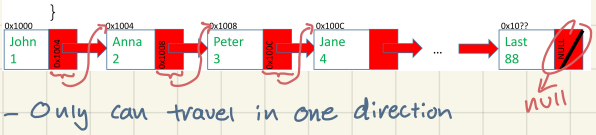
```
#include <stdio.h>
#include <stdlib.h>                                  //include library for malloc() and free()

int main(void)
{
      int i;
      double* item;                                  //declare two pointers
      char* string;
      item = (double *) malloc(10*sizeof(double));   //dynamically memory allocation for 10 elements each
      string = (char *) malloc(10*sizeof(char));

      for(i=0;i<10;i++)                              //Read 10 floating numbers
      {
            scanf("%lf",&item[i]);
      }
      scanf("%*c");                                  //skip the last '\n'

      i=0;                                           //Read 9 character + null character to stop
      char *stringP=string;
      while(i++<9)
            scanf("%c",stringP++);
      *stringP='\0';
      printf("%s\n",string);                         //Print the string

      double* itemP=item;                            //Print the numbers in item
      for(i=0;i<10;i++,itemP++)
            printf("%.2lf ",*itemP);
      printf("\n");

      free(item);                                    //free the allocated memory
      free(string);
      return 0;
}
```

→ Chapter 2

Linked List

Singly-linked list

```
struct node{
        char Name[15];
        int matricNo;
        struct node *nextPtr;  //link
}
```



- Only can travel in one direction
- If loose the pointer, all the other data points are gone

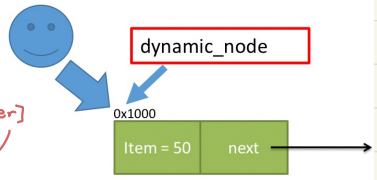
DEFINE AND CREATE A LINKED LIST

```
#include <stdio.h>
#include <stdlib.h>

struct _listnode
{
      int item;   (data)
      struct _listnode *next;
};
typedef struct _listnode ListNode;

int main(void)
{
//static node
      ListNode static_node;
      static_node.data = 50;
      static_node.next = NULL;

//dynamic node
      ListNode* dynamic_node=malloc(sizeof(ListNode));
      dynamic_node->data = 50;
      dynamic_node->next = NULL;

      ListNode* head = dynamic_node;
      free(dynamic_node);

      return 0;
}
```

Create 1st point [Header]

assign pointer to head

- Create a head
  - ListNode* head;
- Multiple ListNode pointers can be created but the node just need to free once in the end



What is head after line 24?

Creating Malloc

```c
typedef struct node{
     int item; struct node *next;
} ListNode;

int main(){
     ListNode *head = NULL, *temp;
     int i = 0;

     while (scanf("%d", &i)){
          if (head == NULL){
               head = malloc(sizeof(ListNode));
               temp = head;
          }
          else{
               temp->next = malloc(sizeof(ListNode));
               temp = temp->next;
          }
          temp->item = i;
     }
     temp->next = NULL;
     return 0;
}
```

while user input integer

allocate next in the chain

cast the pointer by (ListNode*)

allocate user input i

add free(head); free(temp);

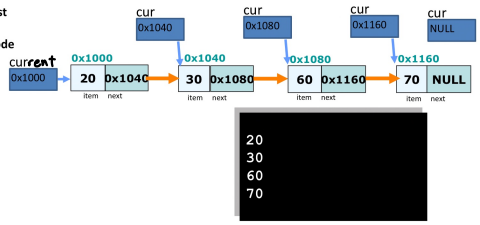
Printing Linked Lists

DISPLAY: printList()

```c
void printList(ListNode *cur);
```

1. Given the head pointer of the linked list
2. Print all items in the linked list
3. From first node to the last node

```c
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```



Finding Linked Lists

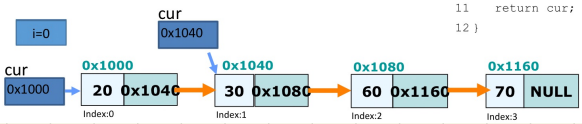
SEARCH: findNode()

```c
ListNode* findNode(ListNode *cur, int i);
```

Looking for the 1st node in the list

1. Given the head pointer of the linked list and index *i=1*

```c
ListNode *findNode(ListNode* c
{
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```



Inserting New Nodes

1) Middle 2) Back 3) Front

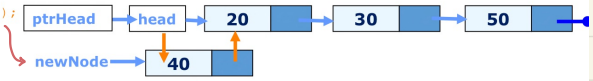
insertNode()

**ptrHead is because of this line

Front

i=0 item=40

```c
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    // If empty list or inserting first node, update head pointer
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    // Find the nodes before and at the target position
    // Create a new node and reconnect the links
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```



For back & middle

Size o List

SIZE: sizeList()

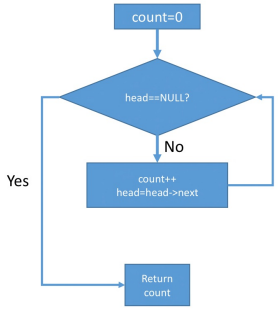
```c
int sizeList(ListNode *head);
```

Given

- the head pointer of the linked list

Return the number of nodes in the linked list

```c
int sizeList(ListNode *head){

    int count = 0;

    while (head != NULL){
        count++;
        head = head->next;
    }

    return count;
}
```



→ Chap 4

Analysis of Algorithm

Time Complexity

1) Count the number of primitive operations

Not Dependent on input

- Declaration: int x;
- Assignment: x =1;
- Arithmetic operations: +, -, *, /, % etc.
- Logic operations: ==, !=, >, <, &&, ||

Dependent on input

i. Repetition Structure: for-loop, while-loop
ii. Selection Structure: if/else statement, switch-case statement
iii. Recursive functions

2) Express in terms of problem size

2 input vs 1 million inputs

★ find rs between function time for different inputs

```
1: j ← 1                          ····· c0
2: factorial ← 1                  ····· c1
3: while j ≤ n do
4:     factorial ← factorial * j  ····· c2   } n iterations ➡ n(c2+c3)
5:     j ← j + 1                  ····· c3   }
```

$f(n) = c_0+c_1+n(c_2+c_3)$

The function increases linearly with n (problem size)

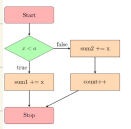
```
1: for j ← 1, m do
2:       for k ← 1, n do
3:             sum ← sum + M[j][k]  ····· c1   } n iterations  n(c1)   } m iterations  m(n(c1))
```

The function increases quadratically with n if m==n

$n' = 10n$ → new n x10 bigger

$f(n') = 10^2 f(n)$

ii. Selection Structure: if/else statement, switch-case statement




```
1: if(x<a)
2:       sum1 += x;
3: else {
4:       sum2 += x;
5:       count ++;
6:       }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis
2. Worst-case analysis
3. Average-case analysis

Selection Structure: switch-case statement

```
1: switch(choice){
2:     case 1: compute the summation; break;   ····· 5n
3:     case 2: search BST; break;              ····· 6log2 n
4:     case 3: print BST; break;               ····· 3n
5:     case 4: search for the minimum; break;  ····· 4 log2 n
6: }
```

Time Complexity

1. Best-case analysis ····· $C + 4\log_2 n$
2. Worst-case analysis ····· $C + 5n$
3. Average-case analysis ····· $C + \sum_{i=1}^{4} p(i)T_i$

Recursive functions

- Count the number of primitive operations in the algorithm
  - Primitive operations in each recursive call
  - Number of recursive calls

```
1 int factorial (int n)
2 {
3     if(n==1) return 1;          ····· c2
4     else return n*factorial(n-1); ····· c1
5 }
```

- *n-1* recursive calls with the cost of c_1.
- The cost of the last call (n==1) is c_2.
- Thus, $c_1(n-1)+c_2$
- It is a linear function

Recursive functions

Count the number of array[0]==a in the algorithm

- array[0]==a in each recursive call
- Number of recursive calls: n-1

```
1  int count (int array[],int n, int a)
2  {
3      if(n==1)
4          if(array[0]==a)
5              return 1;
6          else return 0;
7      if(array[0]==a)
8          return 1+ count(&array[1], n-1, a);
9      else
10         return count (&array[1], n-1, a);
11 }
```

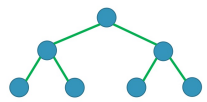
$W_1 = 1$
$W_n = 1 + W_{n-1}$
$= 1 + 1 + W_{n-2}$

$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-\ldots) \cdot (n-(n-1))$

. Recursive functions

- Count the number of multiplication operations in the algorithm

```
preorder (simple_t* tree)
{
   if(tree != NULL){
      tree->item *= 10;
      preorder (tree->left);
      preorder (tree->right);
   }
}
```



Geometric Series:

$$S_n = a + ar + ar^2 + \ldots + ar^{n-1}$$
$$rS_n = ar + ar^2 + \ldots + ar^{n-1} + ar^n$$
$$(1-r)S_n = a - ar^n$$
$$S_n = \frac{a(1-r^n)}{1-r}$$

Prove the hypothesis can be done by mathematical induction

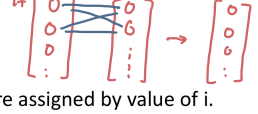
It is known as a **method of forward substitutions**

$W_0 = 0$
$W_1 = 1$
$W_2 = 1 + W_1 + W_1 = 3$
$W_3 = 1 + W_2 + W_2$
$= 1 + 2(1 + W_1 + W_1)$
$= 1 + 2(1 + 2)$
$= 1 + 2 + 4 = 7$
$W_{k-1} = 1 + 2 \cdot W_{k-2}$
$= 1 + 2 + 4 + 8 + \ldots + 2^{k-2}$
$W_k = 1 + 2 \cdot W_{k-1} = 1+2+4+8+\ldots+2^{k-1}$
$= \frac{1-2^k}{1-2} = 2^k - 1$

Cubic Time Complexity

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

```
1     for (i=1; i<=n; i++)
2         M[i] = 0;
3         for (j=i; j>0; j--)
4             for (k=i; k>0; k--)
5                 M[i] += A[j]*B[k];
```

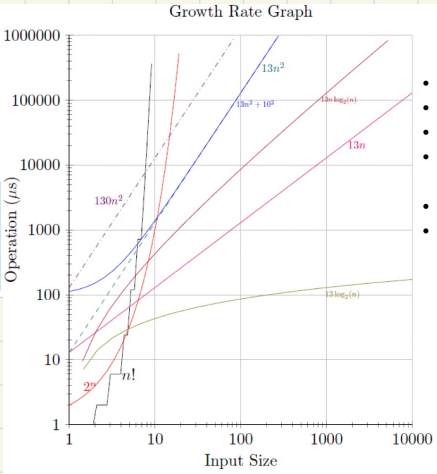


In each outer loop, both j and k are assigned by value of i.
Inner loops takes i^2 iterations
The overall number of iterations is

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2 = \frac{2n^3+3n^2+n}{6}$$
$$= \frac{n(n+1)(2n+1)}{6}$$
$$= \Theta(n^3)$$

Order of Growth

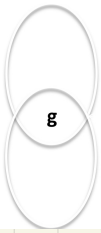
Algorithm	1	2	3	4	5	6
Operation (μsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n
Problem size (n)						
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4x10^{16}$years
10^4	.13	.173	22 mins	3.61hrs	22mins	
10^6	13	259	150 days	1505 days	150days	



- n! is the fastest growth
- 2^n is the second
- 13n is linear
- $13\log_2 n$ is the slowest

- 10^2 can be ignored when n is large
- $13n^2$ and $130n^2$ have similar growth.
  - $130n^2$ slightly faster

Big-Oh (**O**) , Big-Omega (**Ω**) and Big-Theta (**Θ**) are asymptotic (set) notations used for describing the order of growth of a given function.



$f \in \Omega(g)$ Set of functions that grow at higher or same rate as **g**

$f \in \Theta(g)$ Set of functions that grow at same rate as **g**

$f \in O(g)$ Set of functions that grow at lower or same rate as **g**

Big-O is more universal than Θ

Big-Oh Notation (O) – Alternative definition

Definition 3.2 *O-notation: Let f and g be two functions such that $f(n): \mathbb{N} \to \mathbb{R}^+$ and $g(n): \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.*

f(n) = 4n + 3 and g(n) = n

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3n}{n} = 4 < \infty$$

$f(n) = O(g(n))$ $i.e. 4n + 3 \in O(n)$

f(n) = 4n + 3 and g(n) = n^3

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3n}{n^3} = 0 < \infty$$

$f(n) = O(g(n))$ $i.e. 4n + 3 \in O(n^3)$

Time Complexity of Sequential Search

```
pt=head;
while (pt->key != a){
  pt = pt->next;
  if(pt == NULL) break;
}
```

c_1

c_2 (n-1) iterations

Assume that the search key ***a*** is always in the list

1. Best-case analysis: c_1 when **a** is the first item in the list => Θ (1)
2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ => Θ (n)
3. Average-case analysis
   - Assumed that every item in the list has an equal probability as a search key

$$\frac{1}{n}[c_1 + (c_1 + c_2) + (c_1 + 2c_2) + \cdots + (c_1 + (n-1)c_2)] = \frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1))$$
$$= \frac{1}{n}[nc_1 + c_2\sum_{i=1}^{n}(i-1)]$$
$$= c_1 + \frac{c_2}{n}\cdot\frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2}$$

If the search key, a, is not in the list, then the time complexity is $c_1 + nc_2 = \Theta(n)$

Since the probability of the search key is in the list is unknown, we only can have

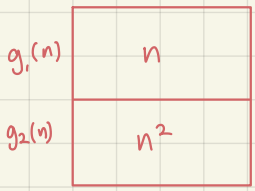
$$T(n) = P(a \text{ in the list})(c_1 + \frac{c_2(n-1)}{2}) + (1 - P(a \text{ in the list}))(c_1 + nc_2)$$

Hence, it is a linear function. **Θ (n)**

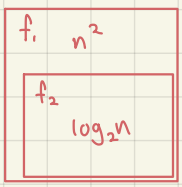
$\lim_{n\to\infty} \frac{f(n)}{g(n)}$	f(n) ∈ O(g(n))	f(n) ∈ Ω(g(n))	f(n) ∈ Θ(g(n))
0	✓		
$0 < C < \infty$	✓	✓	✓
∞		✓	

Simplification Rules for Asymptotic Analysis

1. If $f(n) = O(cg(n))$ for any constant c > 0, then $f(n) = O(g(n))$
2. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
   e.g. $f(n) = 2n$, $g(n) = n^2$, $h(n) = n^3$
3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
   e.g. $5n + 3\log_2 n = O(n)$
4. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$
   then $f_1(n)f_2(n) = O(g_1(n)g_2(n))$
   e.g. $f_1(n) = 3n^2 = O(n^2)$, $f_2(n) = \log_2 n = O(\log_2 n)$
   Then $3n^2 \log_2 n = O(n^2 \log_2 n)$



$f_1(n) + f_2(n)$
$= O(n) + O(n^2)$
$= O(n^2)$



in a loop, total for the loop will be $n^2 \log_2 n$

Order of Growth	Class	Example
1	Constant	Finding midpoint of an array
$\log_2 n$	Logarithmic	Binary Search
n	Linear	Linear Search
$n\log_2 n$	Linearithmic	Merge Sort
n^2	Quadratic	Insertion Sort
n^3	Cubic	Matrix Inversion (Gauss-Jordan Elimination)
2^n	Exponential	The Tower of Hanoi Problem
n!	Factorial	Travelling Salesman Problem

Space Complexity

- Space requirements for an array of n integers - Θ(n)
- If a matrix is used to store edge information of a graph,

  i.e. G[x][y] = 1 if there exists an edge from x to y,

  space requirement for a graph with n vertices is $\Theta(n^2)$

Space/time tradeoff principle

- Reduction in time can be achieved by sacrificing space and vice-versa.

Stack & Queue

Only access items at the top, and read from the top only

Put in items from top, read from bottom

Stack

1. ~~Display: `printStack()`~~
2. Retrieve : `peek()`
3. Insert: `push()`
4. Delete: `pop()`
5. Size: `isEmptyStack()`

Queue

1. ~~Display: `printQueue()`~~
2. Retrieve: `getFront()`
3. Insert: `enqueue()`
4. Delete: `dequeue()`
5. Size: `isEmptyQueue()`

* for efficiency, have a tail pointer

Classic Problems

Stack

- Balanced Parentheses Problem (In lab)
- Algebraic expression conversion (infix, prefix and postfix)
- Recursive functions to Iterative functions

Queue

- Palindromes
- Scheduling in multitasking, network, job, mailbox etc.

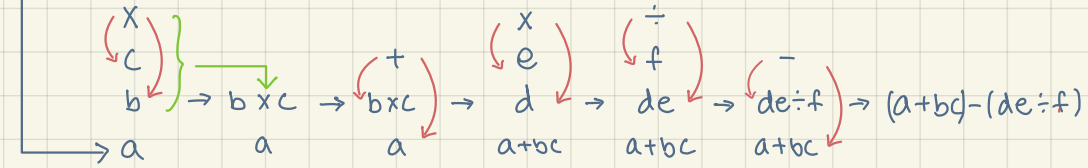
Transfer & Conquer

infix operator

a + b x c − d x e ÷ f

postfix operator

a b c x + d e x f ÷ −



Infix → Postfix

Algorithm 1 Infix Expression to Postfix Expression

```
function IN2POST(String infix, String postfix)
    create a Stack S
    for each character c in infix do
        if c is an operand then                          || If operand (a, b, c...)
            postfix ← c
        else if c = ')' then
            while peek(S) ≠ '(' do
                postfix ← pop(S)
            pop(S)
        else if c = '(' then
            push(c,S)
        else                                             ▷ c is an operator or left parenthesis
            while S ≠ empty && peek(S) != '(' && precedence of peek(S) ≥ precedence of c do
                postfix ← pop(S)                         // Check nxt stack operator precedence
            push(c,S)
    while S is not empty do
        postfix ← pop(S)
```

infix

$a + b \times c - d \times (e \div f)$

S

→ Brackets not necessary in postfix

postfix

$abc \times +def \div \times -$

Postfix → Solution

Algorithm 2 Evaluation Postfix Expression

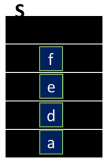
```
function ExePost(String postfix)
    create a Stack S
    for each character c in postfix do
        if c is an operand then
            push(c, S)
        else
            operand1 ← pop(S)
            operand2 ← pop(S)
            result ← Evaluate(operand2, c, operand1)
            push(result, S)
```

$a + b \times c - d \times (e \div f)$

Transform

$abc \times + def \div \times -$

S



$-$ | $d \times (e \div f)$

linear evaluation

→ chap 5a

$a + b \times c - d \times (e \div f)$

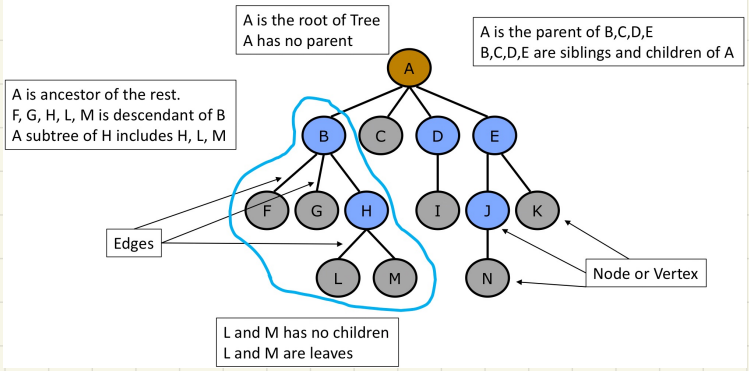
$- + a \times bc \times d \div ef$

x d e÷f first

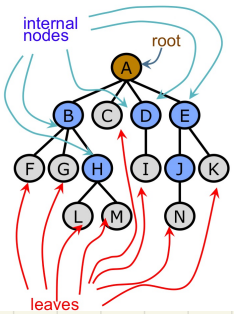
- The expression is known as prefix expression a.k.a Polish notation
- Under this convention, operators appears before its operands
  <operator> <operand> <operand>

Hint: Its algorithm is similar to the postfix expression's.

Trees !

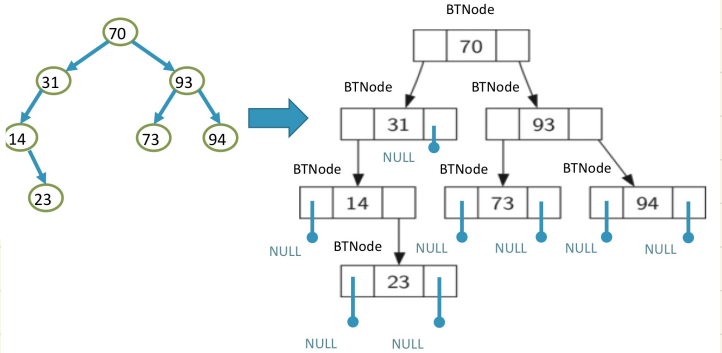


- A tree is composed of nodes
- Types of nodes
  - Root: only one in a tree, has no parent.
  - Internal node(non-leaf): Nodes with children are called internal nodes
  - Leaf (External Node): nodes without children are called leaves



Binary Tree Structure

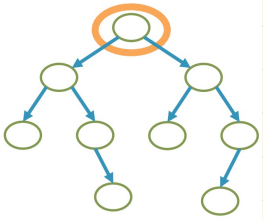
```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```



Traversal

Pseudocode of Binary Tree Traversal

```
TreeTraversal(Node N):
    Visit N;
    If (N has left child)
        TreeTraversal(LeftChild);
    If (N has right child)
        TreeTraversal(RightChild);
    Return; // return to parent
```

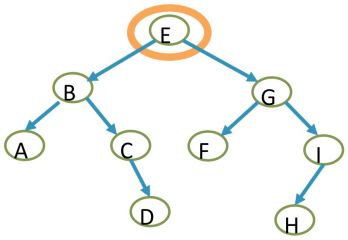


This traversal approach is known as pre-order depth first traversal.

Depth first approach

3 approach to traversal w. recursion

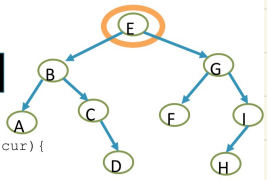
- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- In-order
  - Visit the left child subtree
  - Process the current node's data
  - Visit the right child subtree
- **Post-order**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
  - **Process the current node's data**



Tree Traversal Pre-order: Print

Output:

E B A C D G F I H

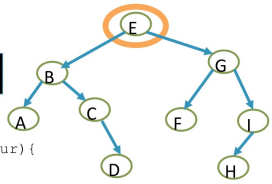



```
void TreeTraversal_pre(BTNode *cur){
    if (cur == NULL)
        return;

    printf("%c  ",cur->item);

    TreeTraversal_pre(cur->left); //Visit the left child node
    TreeTraversal_pre(cur->right);//Visit the right child node
}
```

Tree Traversal In-order: print

Output:

A B C D E F G H I

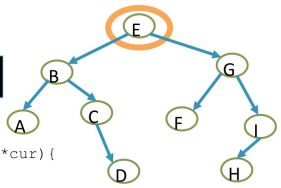



```
void TreeTraversal_in(BTNode *cur){
    if (cur == NULL)
        return;

    TreeTraversal_in(cur->left); //Visit the left child node
    printf("%c  ",cur->item);
    TreeTraversal_in(cur->right);//Visit the right child node
}
```

Tree Traversal Post-order: print

Output:

A D C B F H I G E



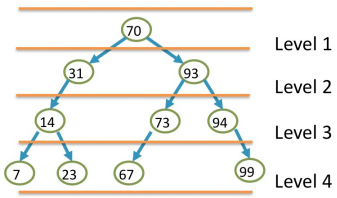

```
void TreeTraversal_post(BTNode *cur){
    if (cur == NULL)
        return;

    TreeTraversal_post(cur->left); //Visit the left child node
    TreeTraversal_post(cur->right);//Visit the right child node
    printf("%c  ",cur->item);
}
```

Breadth first approach

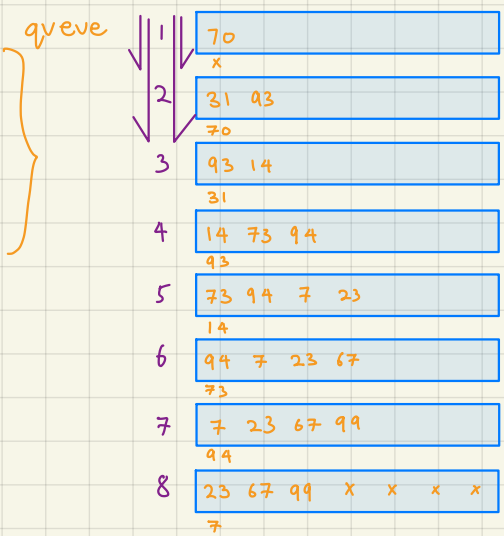
Breadth-First Traversal: Level-by-level

Level-By-Level Traversal:

- Visiting the node
- Remember all its children
  - Use a queue (FIFO structure)



1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

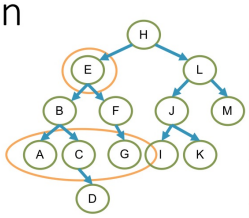



```
void BFT(BTNode *root){
    Queue *q;
    BTNode* node;
    if(root){
       enqueue(q,root); //data type of item in queue is BTNode*
       while(!isEmptyQueue(*q)){
            node = getFront(*q);dequeue(q);
            if(node->left) enqueue(q,node->left);
            if(node->right) enqueue(q,node->right);
       }
    }
}
```

Example: Pre order

Find the k-level Grandchildren

- Given a node X, find all the nodes that are X's grandchildren
- Given node E, we should return grandchild nodes A, C, and G
- What if we want to find k-level grandchildren?
  - **Need a way to keep track of how many levels down we've gone**



2-level grandchildren

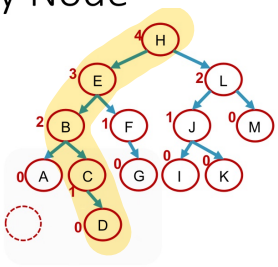
X->left->left
X->left->right
X->right->left
X->right->right

```
1. void findgrandchildren(BTNode *cur, int c){
2.      if (cur == NULL) return;
3.      if (c == k){
4.         printf("%d ", cur->item);
5.         return;
6.      }
7.      if (c < k){
8.         findgrandchildren(cur->left, c+1);
9.         findgrandchildren(cur->right, c+1);
10.}
```

Calculate Height of Every Node

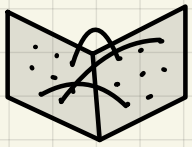
We want each node to report its height

- Leaf node must report 0
- At "null" condition, must report -1




```
int TreeTraversal(BTNode *cur){
    if(cur == NULL)
       return -1;
    int l = TreeTraversal(cur->left);
    int r = TreeTraversal(cur->right);
    int c = max (l, r) + 1;
    return c;
}
```

Search Complexity

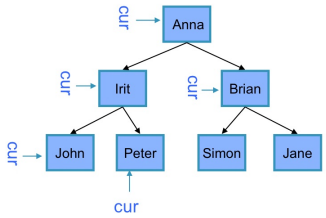


Linear Search by a binary tree

inefficient

- Given a binary tree of names, how do we check whether a given name(e.g., Brian) is in the tree?
- **Use the TreeTraversal (Pre-order) template, to check every node**

How do we insert data into the binary tree?



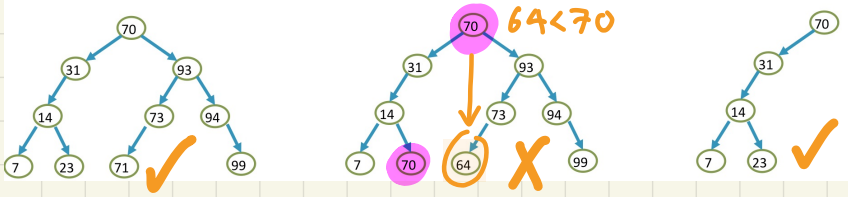

```
TreeTraversal(Node N)
 if N==NULL return;
 if N.item=given_name return;
 TreeTraversal(LeftChild);
 TreeTraversal(RightChild);
 Return;
```

How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+...+7)/7=4 nodes => Θ(n)

Binary Search Tree

If the given binary tree is a binary search tree (BST), then each node in the tree satisfies the following properties:

1. **Node's value is greater than all values in its left subtree.**
2. **Node's value is less than all values in its right subtree.**
3. **Both subtrees of the node are also binary search trees.**



Binary Search Tree: Search

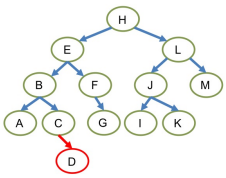
- The approach is a decrease-and-conquer approach
- A problem is divided into two smaller and similar sub-problem, one of which does not even have to be solved
- The method uses the information of the order to reduce the search space.

```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }
    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }
    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

```
void TreeTraversal_pre(BTNode *cur){
    if (cur == NULL) return;
    printf("%c  ",cur->item);
    TreeTraversal_pre(cur->left);
    TreeTraversal_pre(cur->right);
}
```

Binary Search Tree: Insertion

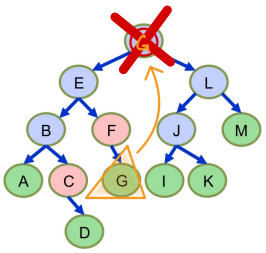
- **After insert a node, the BST must remain as a BST**
- **A duplicate node is not allowed for insertion**
- **A unique position of the given BST will be given to the node**



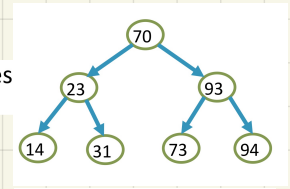

```
BTNode* insertBSTNode(BTNode* cur, char c)
{
    if (cur == NULL){
    BTNode* node = (BTNode*) malloc(sizeof(BTNode));
    node->item = c;
    node->left = node->right = NULL;
    return node;
    }
    if (c < cur->item)
        cur->left  = insertBSTNode (cur->left, c);
    else if (c > cur->item)
        cur->right = insertBSTNode (cur->right, c);
    return cur;
}
```

Binary Search Tree: Deletion

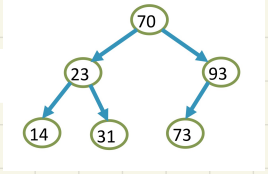
- **After remove a node X, the BST must remain as a BST**
- **Deletion operation on a BST is a bit tricky**
- **Three cases to be considered:**
  1. **X has no children**
     - Remove **X**
  2. **X has one child**
     - Replace X with Y
     - Remove X
  3. **X has two children**
     - **Swap x with successor**
       - the (largest) rightmost node in left subtree
       - the (smallest) leftmost node in right subtree
     - **Perform case 1 or 2 to remove it**



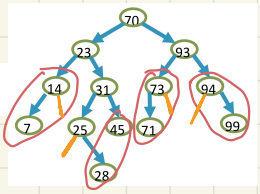
Full Binary Tree: A binary tree of height *H* with no missing nodes. All leaves are at level *H* and all other nodes each have two children



Complete Binary Tree: A binary tree of height *H* that is full to level *H-1* and has level *H* filled in from left to right



Balanced Binary Tree: A binary tree in which the left and right subtrees of any node have heights that differ by at most 1



- The Depth/Level of a node: The number of edges from the node to the root of its tree.

For a complete binary tree with height *H*, we have:

$2^H - 1 < n \leq 2^{H+1} - 1$

where *n* is an integer and the size of the tree

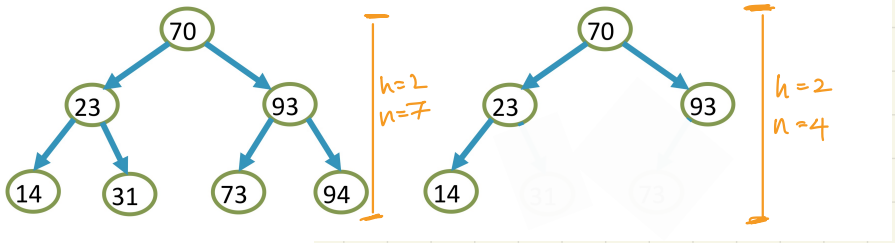
$2^H \leq n < 2^{H+1}$ (eg. $7 < n \leq 15 \equiv 8 \leq n < 16$)

$H \leq \log_2 n < H+1$

If H is an integer, H+1 must be the next integer.

Minimal Height = $\lfloor \log_2 n \rfloor$

Tested!



Binary Search – Worst Case Time complexity

- Assumed that it is a complete binary tree

```
BTNode* findBSTNode(BTNode *cur, char c){
    if (cur == NULL) {
       printf("can't find!");
       return cur;
    }

    if(c==cur->item){
       printf("Found!\n");
       return cur;
    }

    if(c<cur->item)
       return findBSTNode(cur->left,c);
    else
       return findBSTNode(cur->right,c);
}
```

T(n)

Constant c

$T((n-1)/2)$

$T((n-1)/2)$

$$T(n) = T\left(\frac{n-1}{2}\right) + c$$

$$= T\left(\frac{\left(\frac{n-1}{2}\right)-1}{2}\right) + 2c = T\left(\frac{n-1-2}{2^2}\right) + 2c$$

$$= T\left(\frac{\frac{n-1-2}{2^2}-1}{2}\right) + 3c = T\left(\frac{n-1-2-2^2}{2^3}\right) + 3c$$

...

$$= T\left(\frac{n-(1+2\ldots+2^{k-2}+2^{k-1})}{2^k}\right) + kc$$

$$= T\left(\frac{n-2^k+1}{2^k}\right) + kc$$

$$0 < \frac{n-2^k+1}{2^k} \leq 1$$

$$0 < \frac{n+1}{2^k} - 1 \leq 1$$

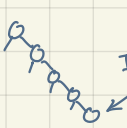
$$1 < \frac{n+1}{2^k} \leq 2$$

$$2^k < n+1 \leq 2^{k+1}$$

$$k < \log_2(n+1) \leq k+1$$

$$\lceil \log_2(n+1) \rceil = k+1$$

Ceiling function: largest time:



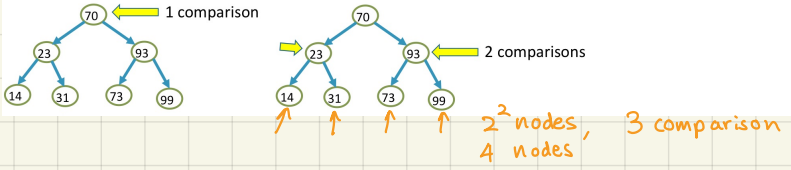
If at this location, time complexity is $\Theta(\log_2 n)$ because have to touch on every node

Binary Search – Average Case Time Complexity

- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search (worst case): $\Theta(\log_2 n)$

$$A(n) = qA_s(n) + (1-q)A_f(n)$$

For $A_s(n)$, we assume $n = 2^k - 1$ first



2^2 nodes, 3 comparison
4 nodes

$$A(n) = qA_s(n) + (1-q)A_f(n)$$

- For $A_s(n)$, we assume $n = 2^k - 1$ first
- We can observe that:
  - 1 position requires 1 comparison
  - 2 positions requires 2 comparisons
  - 4 positions requires 3 comparisons
  - … …
  - 2^{t-1} positions requires t comparisons
- n=2^k-1, we have

$$A_s(n) = \frac{1}{n}\sum_{t=1}^{k} t2^{t-1}$$
$$= \frac{(k-1)2^k + 1}{n}$$
$$= \frac{[\log_2(n+1) - 1](n+1) + 1}{n}$$
$$= \log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}$$

$$\sum_{t=1}^{k} t2^{t-1} = 1\cdot 1 + 2\cdot 2 + 3\cdot 4 + 4\cdot 8 + \ldots + k\cdot 2^{k-1}$$
$$2\sum_{t=1}^{k} t2^{t-1} = \quad 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + (k-1)\cdot 2^{k-1} + k\cdot 2^k$$
$$(2-1)\sum_{t=1}^{k} t2^{t-1} = -1\cdot 1 - 1\cdot 2 - 1\cdot 4 - 1\cdot 8 - \ldots - 1\cdot 2^{k-1} + k\cdot 2^k \quad \triangleright \text{ eq. } 2 - \text{eq. } 1$$
$$\sum_{t=1}^{k} t2^{t-1} = -2^k + 1 + k\cdot 2^k \quad \triangleright \text{ geometric series}$$
$$= 2^k(k-1) + 1$$

- The time complexity is

found, fail, Probability fail

$$A_q(n) = qA_s(n) + (1-q)A_f(n)$$
$$= q[\log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}] + (1-q)(\log_2(n+1))$$
$$= \log_2(n+1) - q + q\frac{\log_2(n+1)}{n}$$
$$= \Theta(log_2(n))$$

Probability success

$n \to \infty$, $\frac{\log_2(n+1)}{n} \to 0$

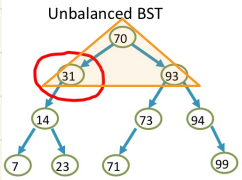
- Binary search does approximately $\log_2(n+1)$ comparisons on average for n entries.
  - q is probability which is always ≤ 1
  - $\frac{\log_2(n+1)}{n}$ is very small especially when n >> 1

Tree Balancing

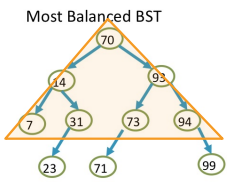
Tree Balancing

- A BST is height-balanced or balanced if the difference in height of both subtrees of any node in the tree is either zero or one.
- Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels

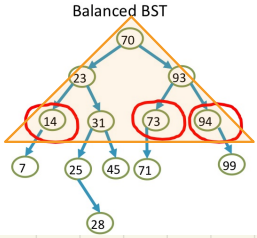
Unbalanced BST



Most Balanced BST



Balanced BST

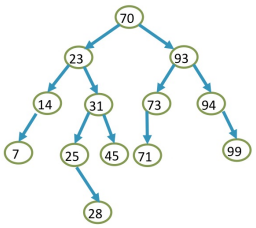


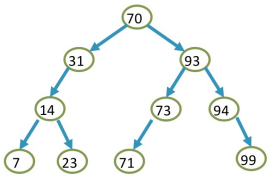
AVL TREE

Tree Balancing (normal)

- How do you balance a binary search tree?
  1. Sort all the data in an array and reconstruct the tree
  2. The AVL Tree:
     - It is a locally balanced tree:
     - Heights of left vs right subtrees differ by at most 1
     - invented by Adel'son-Velskii and Landis in 1962

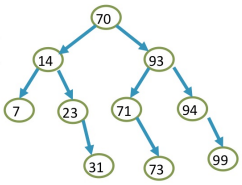
AVL Tree





Θ(n)

7,14,23,31,70,71,73,93,94,99




```
void treeBalance(int data[], int first, int last,
BTNode *root)
{
   if(last>=first){
    int middle = (first+last)/2;
    insertBSTNode(root,data[middle]);
    treeBalance(data,first,middle-1,root->left);
    treeBalance(data,middle+1,last,root->right);
    }
}
```

Θ(n)

Most Balanced BST: Each tree node has exactly two child nodes except for the bottom 2 levels

1) Need extra array to store sorted data
2) Rebuild the whole tree

Time complexity: Θ(nlogn)
Space complexity: Θ(n) (additional array)

- Sort all the data in an array and reconstruct the tree
1. In-order traversal visits every node in the given BST. We obtain the sorted data:

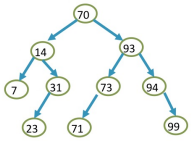
   7,14,23,31,70,71,73,93,94,99
2. Storage it in an array
3. Take the middle element of the array as the root of the tree: 70
4. The first half of the array is used to build the left subtree of 70

   7,14,23,31
5. The second half of the array is used to build the right subtree

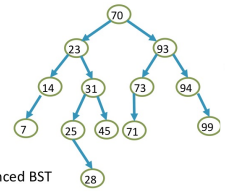
   71,73,93,94,99
6. Step 4 and Step 5 recursively repeat the step 3-5.

AVL Tree

Most Balanced BST



- Add/ remove only one node to/ from a BST
- The BST may become unbalance after insertion or removal
- Instead of reconstructing the BST via sorting data, the BST can be locally balanced
- It is known as AVL Tree
- The height of left and right subtrees of every node differ by at most one

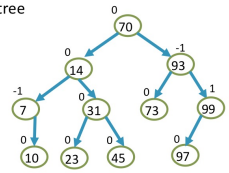


Balanced BST

Balance Factor

- Balance Factor: Height of Left Subtree – Height of Right Subtree
- All the leave have 0 Balance Factor

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
    int height;
} BTNode;
```

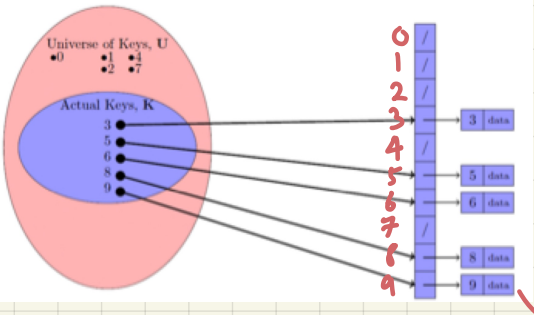


- An AVL tree: The height of left and right subtrees of every node differ by at most one.
  - Balance Factor of each node in an AVL tree can only be -1, 0 or 1.
  - Node insertion or node removal from the tree may change the balance factor of its ancestors (from parent, grandparent, grand-grandparent etc. to the root of the tree)

→ Missing lake lecture

Hash Table

Question: How do we construct a search algo that is of O(1) time complexity?



1) Assign the value to an array where allocated in arr(value) = value

* Time complexity = O(1)

BUT: Size complexity is enormous

Near ∞ size!

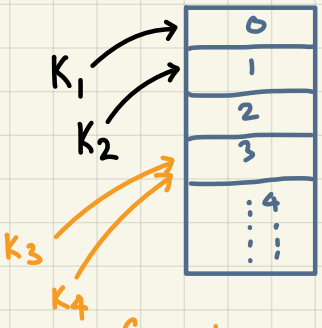
Solve Using Hash Table!

→ Mapping a key (data) to a unique index

→ Hash function is an array of h size

→ For every key (input), call the hash function to map into one of the values in 0 - h-1

→ For n keys: load factor : $\alpha = \frac{n}{h}$



Sometimes, when many key to one hash, collision

Key → Hash f(x) → Array [i]

Hash as "random" as possible

1. Modulo Arithmetic
2. Folding
3. Mid-square
4. Multiplicative Congruential Method
5. Etc.

1. Modulo Arithmetic: $H(k) = k \bmod h$
   - E.g. $h = 13$ & $k = 37699$ → $H(k) = 37699 \bmod 13 = 12$
   - In practice, h should be a prime number, but not too close to any power of 2

2. Folding
   - Partition the key into several parts and combine the parts in a convenient way
   - Shift folding: Divide the key into a few parts and added up these parts
   - X = abc → H(X) = (a + b + c) mod h
   - E.g. $H(123456789) = (123 + 456 + 789) \bmod 13 = 3$

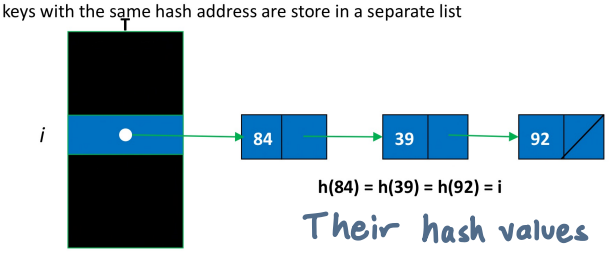
3. Mid-square
   - The key is squared and the middle part of the result is used as the hash address
     - E.g. $k=3121$, $k^2 = 3121^2 = 9740641$ → $H(k) = 406$

4. Multiplicative Congruential Method
   - Pseudo-random number generator
     - $H(k) = (a \times k) \bmod h$
     - E.g. $k = 5$, $a = 6$ → $H(k) = (5 \times 6) \bmod 13 = 4$

Solving Collisions

① Closed Address

- Keys are not stored in the table itself
- All the keys with the same hash address are store in a separate list



Their hash values same!

- During searching, the searched element with hash address i is compared with elements in linked list H[i] sequentially
- In closed address hashing, there will be α number of elements in each linked list on average.
  - ↳ load factor

Worst case : $\Theta(n)$
- All items in one hash functions, become LL traversal

Average case : $\Theta(1+\alpha)$
↳ all elements distributed well
→ Unsuccessful search on average is $\Theta(\alpha)$ as average item overlap is α

② Open Addressing

- Keys are stored in the table itself
- α cannot be greater than 1
- When collision occurs, probe is required for the alternate slot
  - Ideally, the probing approach can visit every possible slot.

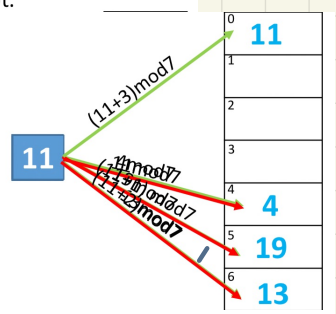
1. Linear Probing: probe the next slot

$H(k,i) = (k+i) \bmod h$ where $i \in [0, h-1]$

eg. $H(k,i) = (k+i) \bmod 7$

$k \in \{4, 13, 19, 11\}$

Primary clustering:
- A long runs of occupied slots
- Average search time is increased



- Sequentially search

2. Quadratic Probing

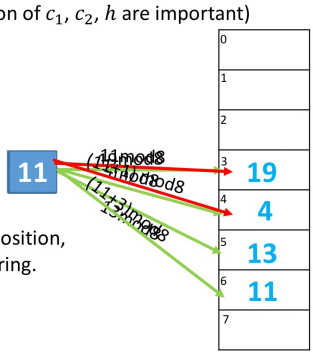
$H(k,i) = (k + c_1 i + c_2 i^2) \bmod h$ where c_1 and c_2 are constants, $c_2 \neq 0$

- May not all hash table slots be on the probe sequence (selection of c_1, c_2, h are important)
  - Any prime number h choice will only have at most h/2 distinct probes
- For $h = 2^n$, a good choice for the constants are $c_1 = c_2 = \frac{1}{2}$

eg. $H(k,i) = \left(k + \frac{1}{2}i + \frac{1}{2}i^2\right) \bmod 8$

$k \in \{4, 13, 19, 11\}$

$(\frac{1}{2}i + \frac{1}{2}i^2) \bmod 8$
1
3
6
2
7
5
4



- Secondary Clustering: if two keys have the same initial probe position, their probe sequences will be the same. This will form a clustering.
  - Inserting k=3 in the previous example.

- Hash input value & repeat number i
- When collide, i++, new hash value

3. Double Hashing: a random probing method

$H(k,i) = (k + iD(k)) \bmod h$ where $i \in [0, h-1]$ and $D(k)$ is another hash function

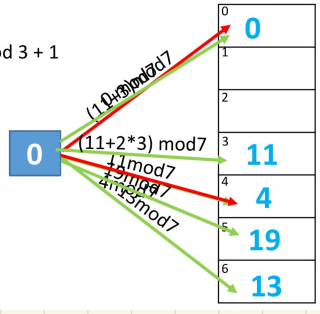
- The hash table size h should be a prime number

eg. $H(k,i) = (k + iD(k)) \bmod 7$

$D(k) = (k) \bmod 3 + 1$

$k \in \{0, 4, 13, 19, 11\}$

D(11) = 11 mod 3 + 1



- Hash input value & repeat number i
- When collide, i++, new hash function add to old hash function

- Closed-address Hashing : Separate Chaining: O(α) on average
- Open-address Hashing: O($f(\frac{1}{1-\alpha})$)

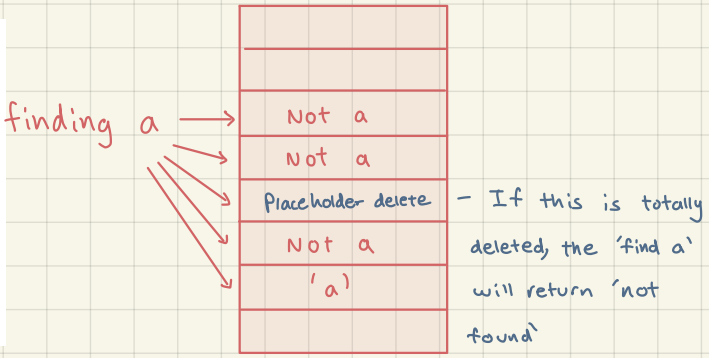
Time Complexity

Linear Probing
- Successful Search: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
- Unsuccessful Search: $\frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$

Double Hashing
- Successful Search: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
- Unsuccessful Search: $\frac{1}{1-\alpha}$

Delete A Key Under Open Addressing

- Leave the deleted key in the table
- Make a marker indicating that it is deleted
- Overwrite it when a new key is inserted to the slot
- May need to do a "garbage collection" when a large number of deletions are done
  - To improve the search time



Rehashing: Expanding the Hash Table

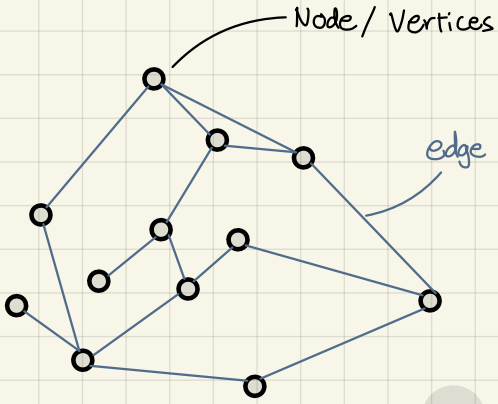
- As α increases, the time complexity also increases

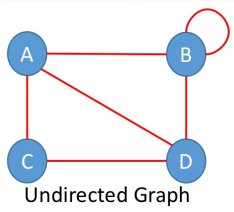
Solution:

- Increase the size of hash table (doubled)
- Rehash all keys into new larger hash table

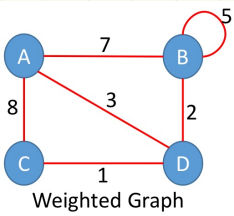
Graph

- A graph $G = (V, E)$ consists of two finite sets:
  - A set V of vertices/ nodes
  - |V| is the number of vertices
  - A set E of edges/arcs/links that connect the vertices
  - $E = \{(x, y) | x, y \in V\}$
  - $|E|$ is the number of edges ranged from 0 to $\frac{|V|(|V|-1)}{2}$
  - Degree of a vertex is the number of edges incident to it
  - A tree is a special graph with no cycle

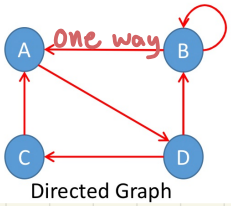




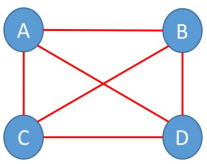
Undirected Graph



Weighted Graph

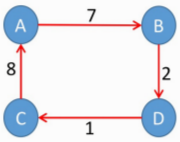


Directed Graph



Complete Graph

- A path is a sequence of distinct vertices, each adjacent to the predecessor (except for the first vertex). |V| = |E|+1
  - ABDC
- A cycle is a path containing at least three vertices such that the last vertex on the path is the same as the first. |V| = |E|
  - ABDCA



* Multi-edge

- If e = (*x*, *y*) is an edge in an undirected graph, then *e* is **incident** with *x* and *y*; *x* is **adjacent** to *y* and vice versa.
- If E is unordered, then G is undirected; otherwise, G is a directed graph.
- If e = (x, y) is an edge in a directed graph, then y can be reached from x through one edge, so target y is adjacent to source x (but it doesn't mean x is adjacent to y).

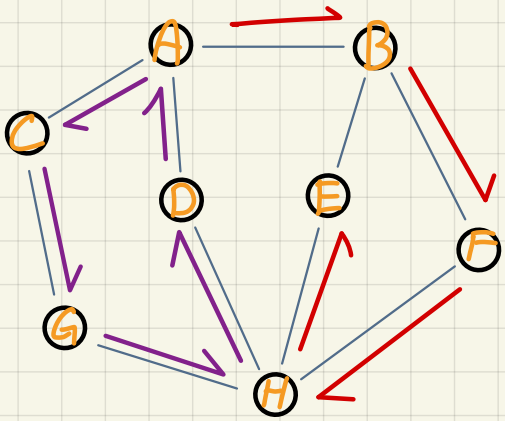
$e = (x, y)$, $x \xrightarrow{e} y$

$E = \{e_1, \ldots\}$ (x, y) (y, x) $\rightarrow$ $x - y$

$e = (x, y)$ $x \rightarrow y$

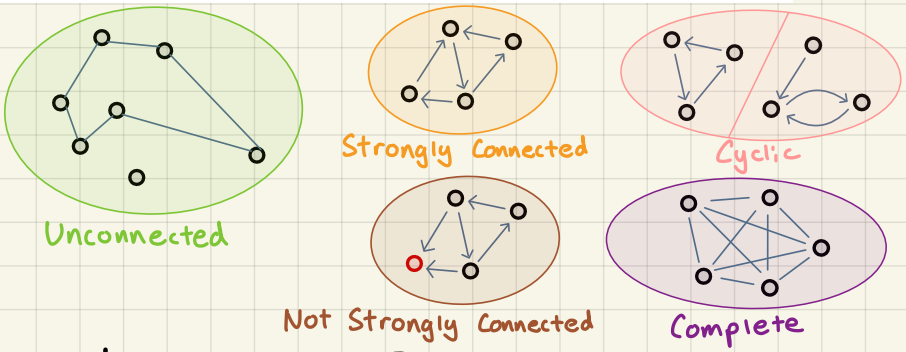
$e \neq (y, x) \rightsquigarrow y \not\rightarrow x$

Path & Cycle



- A path is a sequence of distinct vertices, each adjacent to the predecessor (except for the first vertex). |V| = |E|+1
  <A, B, F, H, E>
- A cycle is a path containing at least three vertices such that the last vertex on the path is the same as the first. |V| = |E|
  <C, G, H, D, A, C>

- An undirected graph is connected if there is a path from any vertex to any other vertex.
- A directed graph is strongly connected if there is a path from any vertex to any other vertex.
- A graph is cyclic if it contains one or more cycles; otherwise it is acyclic.
- A complete graph on n vertices is a simple undirected graph that contains exactly one edge between each pair of distinct vertices.
  - $|E| = \frac{|V|(|V|-1)}{2}$



How to represent?

Adjacency Matrix

- Use a matrix (2-D array) with size $|V| \times |V|$

```
typedef struct _graph{
        int vSize;
        int eSize;
        int **AdjM;
}Graph;
```

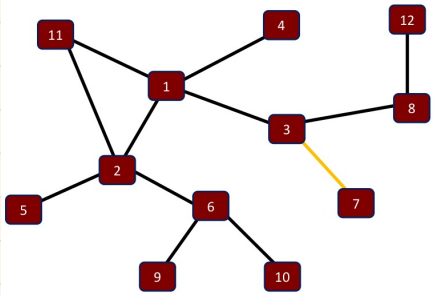
- $(u, v) \in E$ implies AdjM[u][v] = 1; Otherwise AdjM[u][v] = 0.
- If a graph is undirected, then AdjM is symmetric
  - AdjM[u][v] = AdjM[v][u]
- If a graph is directed, then AdjM[u][v] = 1 iff $(u, v) \in E$ but it does not imply $(v, u) \in E$ and AdjM[v][u] = 1.

Adjacency Matrix

```
typedef struct _graph{
        int vSize;
        int eSize;
        int **AdjM;
}Graph;
```

→ Represents single cell, x & y & Content

- access time for AdjM[u][v] is constant
- when graph is sparsely connected, most of the entries in AdjM are zeros



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

Space = $\Theta |V|^2$

Time complexity = O(1)

Adjacency List

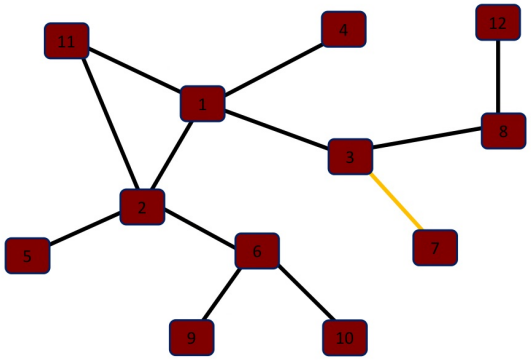
```
struct _listnode
{
        int id; //or weight
        struct _listnode *next;
};
typedef struct _listnode ListNode;
typedef struct _graph{
        int vSize;
        int eSize;
        ListNode **AdjL;
}Graph;
```

- more efficient for space, though

- Use an array to represent the vertices
- For each vertex, use a linked list to represent the connections to other vertices
- Access time for AdjM[u][v] is linear
- Space complexity is lower, O(|V|+|E|)

Adjacency List

- Array size is |V|.
- Total number of nodes in link lists is 2|E|



1	→ 2 → 3 → 4 → 11
2	→ 11 → 1 → 5 → 6
3	→ 1 → 8 → 7
4	→ 1
5	→ 2
6	→ 10 → 9 → 2
7	→ 3
8	→ 12 → 3
9	→ 6
10	→ 6
11	→ 2 → 1
12	→ 8

Space = |V| + |E|

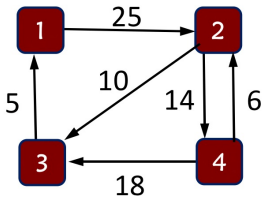
Time complexity = $\Omega(|V|)$

Represent Weighted Graphs

- In the array of adjacency lists, the weight can be stored as a data field in each list node
- In the adjacency matrices, the weight can be stored
  - The element at the *u*-th row and the v-th column can be defined as:

$$AdjM[u][v] = \begin{cases} W(u,v) & \text{if } (u,v) \in E \\ c & \text{otherwise} \end{cases}$$

- Constant c can be defined as 0 (weight as capacity) or some very large number ∞ (weight as cost)



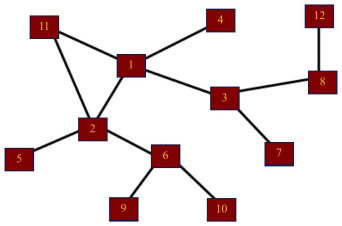
	1	2	3	4
1	0	25	0	0
2	0	0	10	14
3	5	0	0	0
4	0	6	18	0

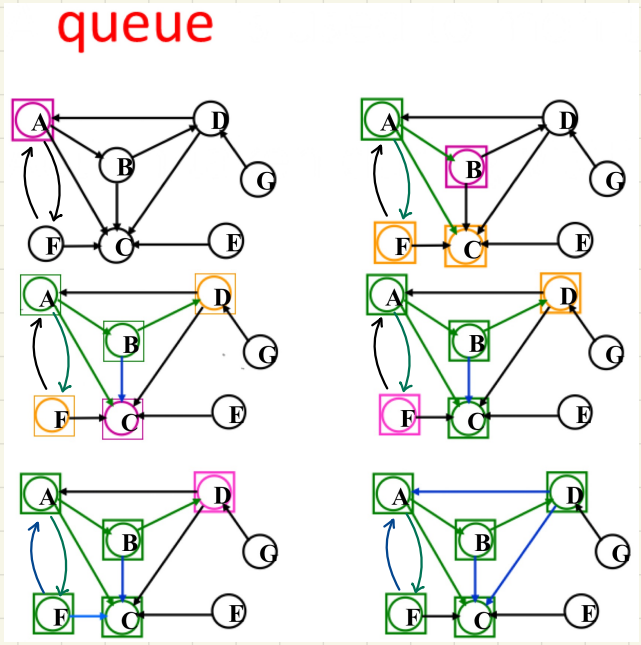
1	→ (2, 25)
2	→ (3, 10) → (4, 14)
3	→ (1, 5)
4	→ (2, 6) → (3, 18)

Graph Traversal

Breadth First Search

- Work similar to level-order traversal of the trees
- BFS systematically explores the edges directly connected to a vertex before visiting vertices further away.





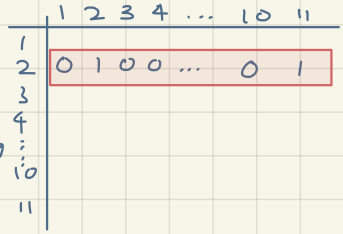
BFS Algorithm

```
function BFS(Graph G, Vertex v)
    create a Queue, Q
    enqueue v into Q
    mark v as visited
    while Q is not empty do
        dequeue a vertex denoted as w
        for each unvisited vertex u adjacent to w do
            mark u as visited
            enqueue u into Q
        end for
    end while
end function
```

- If a vertex has several unmarked neighbours, it would be equally correct to visit them in any order.
- If the shortest path from s to any vertex v is defined as the path with the minimum number of edges, then BFS finds the shortest paths from s to all vertices reachable from s.
- The tree built by BFS is called the **breadth first spanning tree** (when graph G is connected).

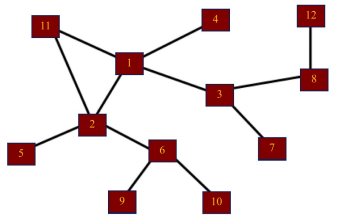
Time Complexity of BFS

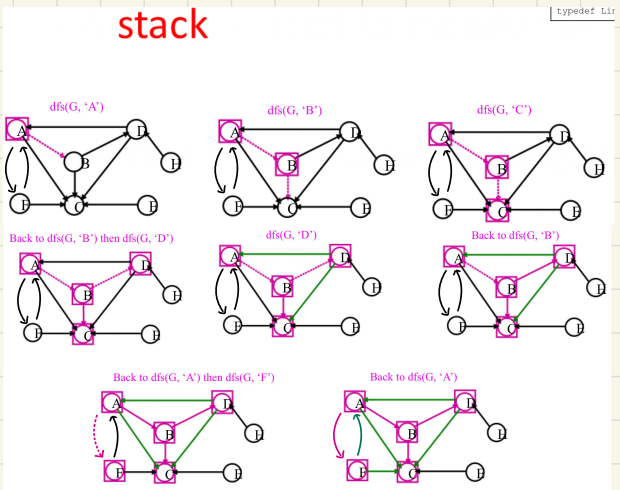
- Each edge is processed once in the while loop for a total cost of Θ(|E|)
- Each vertex is queued and dequeued once for a total cost of Θ(|V|)
- The worst-case time complexity for BFS is
  - Θ(|V| + |E|) if graph is represented by adjacency lists
  - Θ(|V|2) if graph is represented by an adjacency matrix
    - each vertex takes Θ(|V|) to scan for its neighbours



Depth First Search

- Work similar to preorder traversal of the trees
- DFS systematically explores along a path from vertex v as deeply into the graph as possible before backing up.

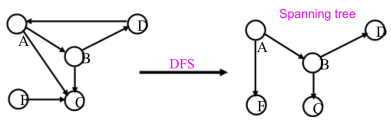




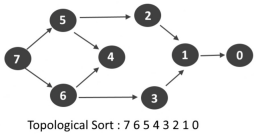
DFS Algorithm

```
function DFS(Graph G, Vertex v)
    create a Stack, S
    push v into S
    mark v as visited
    while S is not empty do
        peek the stack and denote the vertex as w
        if no unvisited vertices are adjacent to w then
            pop a vertex from S
        else
            push an unvisited vertex u adjacent to w
            mark u as visited
        end if
    end while
end function
```

- If the graph is strongly connected, the tree T, constructed by the DFS algorithm is a spanning tree, i.e., a set of |V|-1 edges that connect all vertices of the graph. T is called the **depth first search tree**.



Applications of DFS



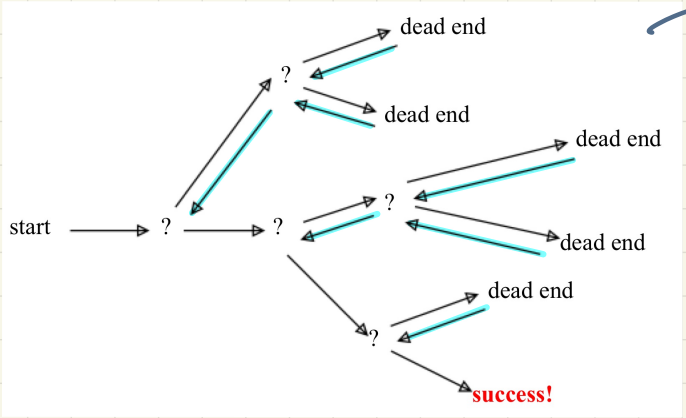
Topological Sort : 7 6 5 4 3 2 1 0

- Topological Sorting
- Finding connected components
- Finding articulation points (cut vertices) of the graph
- Finding strongly connected components
- Solving puzzles

Time Complexity of DFS

- The DFS algorithm visits each node exactly once; every edge is traversed once in forward direction (exploring) and once in backward direction (backtracking).
- Using adjacency-lists, time complexity of DFS is $\Theta(|V| + |E|)$.

Backtracking

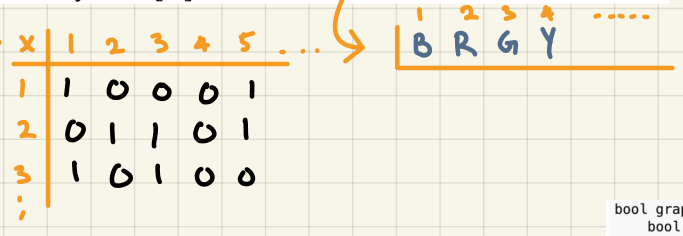


If no more choice, go back until you have a choice

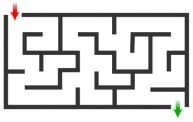
```
Backtracking(N)
    If N is a goal node, return "success"
    Else if N is a leaf node, return "failure"
    For each child C of N,
        If Backtracking(C) == "success"
            Return "success"
    Return "failure"
```

Coloring a map

- You wish to color a map with not more than n colors
- Adjacent areas must be in different colors
- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking
- Input format:
  - 2D adjacency matrix representation of the graph [V][V]
  - Number of colors m
- Output format:
  - array color[V] that should have numbers from 1 to m



Solving a maze



- Given a maze, find a path from start to finish
- At each intersection, you have to decide:
  - Go straight
  - Go left
  - Go right
- You don't have enough information to choose correctly
  - Each choice leads to another set of choices
  - One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

Coloring problem: Backtracking

- Create a recursive function that takes current index
- If the current index is equal to the number of vertices
  - Print the color configuration in output array.
- Assign each color to a vertex (1 to m).
- For every assigned color, check if the configuration is safe, recursively call the function with next index and number of vertices
  - If any recursive function returns true break the loop and return true.
- If no recursive function returns true then return false.

```
bool graphColoringUtil(
    bool graph[V][V], int m,
    int color[], int v)
{
    /* base case: */
    if (v == V)
        return true;

    /* Consider this vertex v and
    try different colors */
    for (int c = 1; c <= m; c++) {
        /* Check if color c to v is fine*/
        if (isSafe(
                v, graph, color, c)) {
            color[v] = c;

            /* recur to assign colors to
            rest of the vertices */
            if (
                graphColoringUtil(
                    graph, m, color, v + 1)
                == true)
                return true;

            /* If c is not successful -> remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned \*/
    return false;
}
```

```
bool isSafe(
    int v, bool graph[V][V],
    int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (
            graph[v][i] && c == color[i])
            return false;
    return true;
}
```

The Eight Queens Problem

- A chessboard has 8 rows and 8 columns
- A queen can move within its row or its column or along its diagonal
- Place 8 queens on the board
  - No queen can attack any other queen in a move

- Exhausting search will take $\binom{64}{8} = 4.43$ billion ways
- Each row can contain exactly one queen: $8! = 40{,}320$
- Better algorithm?

Backtracking Algorithm

1. Starts by placing a queen on the top left corner of the chess board.
2. Places a queen on the second column and moves her until a place where she cannot be hit by the queen on the first column.
3. Places a queen on the third column and moves her until she cannot be hit by either of the first two queens and so on.
4. If there is no place for the i^{th} queen, the program backtracks to move the $(i-1)^{th}$ queen.
5. If the $(i-1)^{th}$ queen is at the end of the column, the program removes the queen and backtracks to the $(i-2)$ column and so on.
6. If the current column is the last column and a safe place has been found for the last queen, then a solution to the puzzle has been found.

```
function NQUEENS(Board[N][N], Column)
    if Column >= N then return true                          ▷ Solution is found
    else
        for i ← 1, N do
            if Board[i][Column] is safe to place then
                Place a queen in the square
                if NQueens(Board[N][N], Column + 1) then return true   ▷ Solution is found
                end if
                Delete the queen
            end if
        end for
    end if
    return false                                             ▷ no solution is found
end function
```

Dynamic Programming

Fibonacci

Dynamic Programming = Recursion + Memoization

- Recursion: problem can be solved recursively
- Memoization: Store optimal solutions to sub-problems in table (or memory or cache)

Fibonacci sequence: recursive algorithm

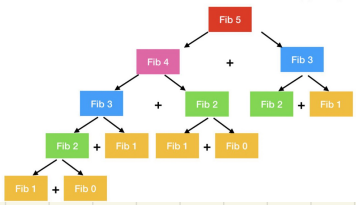
```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

   return Fib(n-1) + Fib(n-2);
}
```

It has a serious issue!

What's the problem?

Many subproblems are overlapping: a lot of recomputation

Complexity: O(2ⁿ)



- It is similar to divide-and-conquer strategy
  - Breaking the big problem into sub-problems
  - Solve the sub-problems recursively
  - Combining the solutions to the sub-problems
- What is the difference between them?
  - DP can be applied when the sub-problems are not independent
    - Every sub-problem is solved once and is saved in a table
  - The problem usually can have multiple optimal solutions
    - DP may just return one of them
- If the sub-problems are independent, DP is not useful!

Dynamic Programming Approaches

- Top-down approach
  - Recursively using the solution to its sub-problems
  - Memoize the solutions to the sub-problems and reuse them later
- Bottom-up approach
  - Figure out the order of calculation
  - Solve the sub-problems to build up solutions to larger problem

Fibonacci: Top-down approach

start @ 5

Store an array M

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

Complexity: O(n)

```
Fib(n)
{
   if (n == 0)
         M[0] = 0; return 0;        } Base case
   if (n == 1)
         M[1] = 1; return 1;

   if (M[n-1] == -1)                //F(n-1) was not calculated
         M[n-1] = Fib(n-1)          //calculate F(n-1) and store in M

   if (M[n-2] == -1)                //F(n-2) was not calculated
         M[n-2] = Fib(n-2)          //calculate F(n-2) and store in M

   M[n] = M[n-1] + M[n-2]
   return M[n];
}
```

Fibonacci: Bottom-up approach

start @ 1

Store an array M

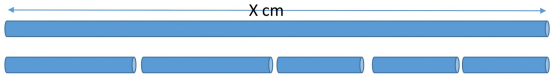
0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

Complexity: O(n)

```
Fib(n)
{
   M[0] = 0;
   M[1] = 1;          } Base Case

   int i = 0;
   for (i = 2; i<=n; i++)
         M[i] = M[i-1] + M[i-2];     } Start from bottom
   return M[n];
}
```

Rod Cutting Problem

Given a rod of a certain length and price of rod of different lengths, determine the maximum revenue obtainable by cutting up the rod at different lengths based on the prices.



Length cm	1	2	3	4	5	6	7	8	9
Price $	1	5	8	9	10	17	17	20	24

Other examples of DP

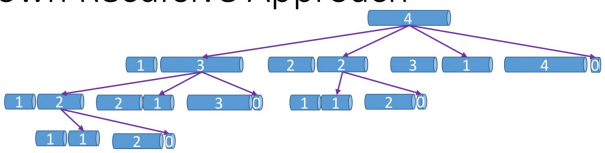
- String algorithms like longest common subsequence, longest increasing subsequence, longest common substring etc.
- Graph algorithms like Bellman-Ford algorithm, Floyd's algorithm
- Chain matrix multiplication

- Rod Cutting
- 0/1 Knapsack
- Travelling salesman problem
- Subset Sum

If a rod of length 4,

Length of each piece	Total Revenue
4	9
1 + 3	1+8 = 9
1 + 1 + 2	1+1+5 =7
1 + 1 + 1 + 1	1+1+1+1=4
2 + 2	**5+5 =10**

Naïve Top-down Recursive Approach




```
Cut-Rod (p,n)
begin
    if n==0
        return 0
    q ← -∞
    for i = 1 to n do
        q ← max (q, p[i] + Cut-Rod(p, n-i))
    return q
end
```

The recursive calls will repeatedly find the revenue for a rod of the same length. Its time complexity is $\Theta(2^n)$

Top-down Memoized Approach

- The result of each sub-problem is stored and reused

```
Cut-Rod (p,n)
begin
        r[1,…,n] ← {0}
        return Mem-Cut-Rod-Aux(p,n,r)
end
```

```
Mem-Cut-Rod-Aux (p,n,r)
begin
        if n==0
                return 0
        if(r[n]>0)
                return r[n]
        else
            q ← -∞
            for i = 1 to n do
                q ← max (q, p[i] + Mem-Cut-Rod-Aux(p, n-i, r))
            r[n] ← q
        return q
end
```

Bottom-up DP Approach

```
DP-Cut-Rod (p,n)
begin
        r[1, …, n] ← {0}
        for j = 1 to n do
            for i = 1 to j do
                r[j] ← max (r[j], p[i] + r[j-i])
        return r[n]
end
```

- The bottom-up and top-down versions has the same asymptotic running time, $\Theta(n^2)$

Length cm	1	2	3	4	5	6	7	8	9
Price $	1	5	8	9	10	17	17	20	24
Max Rev $	1	5	8	10	13	17	18	22	25

0/1 Knapsack

- Given *n* items, where the i[th] item has the size s_i and the value v_i
- Put these items into a knapsack of capacity *C*

- *Optimization problem: Find the largest total value of the items that fits in the knapsack*

$$\max_x \sum_{i=1}^{n} v_i x_i$$

Subject to

$$\sum_{i=1}^{n} s_i x_i \le C$$

$$x_i \in \{0,1\} \qquad i = 1, 2, \dots, n$$

0/1 Knapsack

Subject to

$$\max_x \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} s_i x_i \le C$$

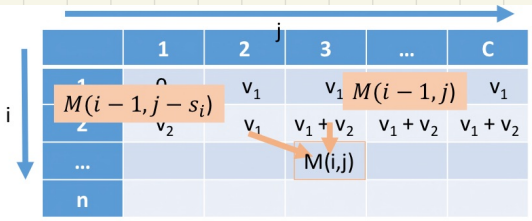
$$x_i \in \{0,1\} \qquad i = 1, 2, \dots, n$$

- Brute-force algorithm
- The i[th] item is either included (1) or excluded (0)
- The time complexity of the algorithm is $\Theta(2^n)$

Item 1	Item 2	Item 3	Value
0	0	0	0
0	0	1	V3
0	1	0	V2
0	1	1	V2+V3
1	0	0	V1
1	0	1	V1+V3
1	1	0	V1+V2
1	1	1	V1+V2+V3

Can you see that some sub-problems are overlapping?

Using DP to solve 0/1 Knapsack



- The recursive formula
  - $M(i,j) = \max\{M(i-1,j),\ M(i-1,j-s_i) + v_i\}$
    - i = 1, … n
    - j = 1, … C

ith item is unused

ith item is used

The capacity of knapsack is 5kg. (C = 5)

Item	Weight s	Value v
1	2kg	$12
2	1kg	$10
3	3kg	$20
4	2kg	$15

Capacity

Item

i \ j	1	2	3	4	5
1	$0	$12	$12	$12	$12
2	$10	$12	$22	$22	$22
3	$10	$12	$22	$30	$32
4	$10	$15	$25	$30	**$37**

Using DP to solve 0/1 Knapsack

- The recursive formula
  - $M(i,j) = \max\{M(i-1,j),\ M(i-1,j-s_i) + v_i\}$
    - i = 1, … n
    - j = 1, … C

ith item is unused

ith item is used

- Create a n-by-C matrix, M
- All the possible sizes from 1 to C
- Bottom up approach
- Time Complexity is $\Theta(nC)$