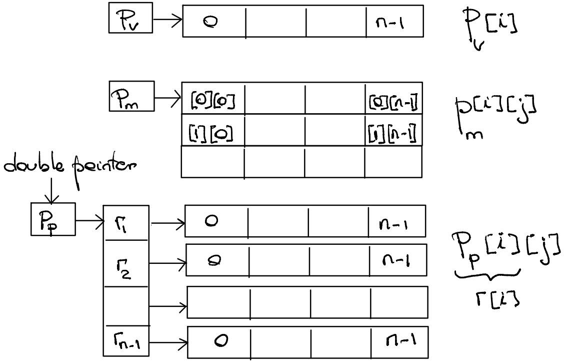
Data Structures

- arrays
- linked lists
- stack
- queues
- binary trees
- forests
- graphs
- hash tables

Arrays



P_v
0
n-1
$P_v[i]$
P_m
[0][0]
[0][n-1]
[1][0]
[1][n-1]
$p_m[i][j]$
double pointer
P_p
r_1
r_2
r_{n-1}
0
n-1
$\underbrace{P_p[i]}_{r[i]}[j]$

C++ Example

```
#include <cstdio>
const int dim = 6;

void afunction()
{
  int a[dim];
  int m[dim][dim/2];
  int* pv = a;
  int(*pm)[dim/2] = m; // NOTICE THE SYNTAX!!!
  int* r[dim];
  int** pp = r;
  for (int i=0; i<dim; i++)
    r[i] = &m[i][0]; // m+i*dim/2
  // initializing matrix
  for (int i=0; i< dim; i++)
    for (int j=0; j<dim/2; j++)
      m[i][j]= (i+1)*10+(j+1);
```

C++ Example

```
/*
int(*pm)[dim/2] = m; // NOTICE THE SYNTAX!!!
int* r[dim];
int** pp = r;
for (int i=0; i<dim; i++)
  r[i] = &m[i][0]; // m+i*dim/2;
*/
printf("printing matrix using pointer to matrix\n");
for (int i=0; i< dim; i++)
  {
  for (int j=0; j<dim/2; j++)
    printf("%d ", pm[i][j]);
  printf("\n");
  }
printf("printing matrix using array of pointers to rows\n");
for (int i=0; i< dim; i++)
  {
  for (int j=0; j<dim/2; j++)
    printf("%d ", pp[i][j]);
  printf("\n");
  }
}
```

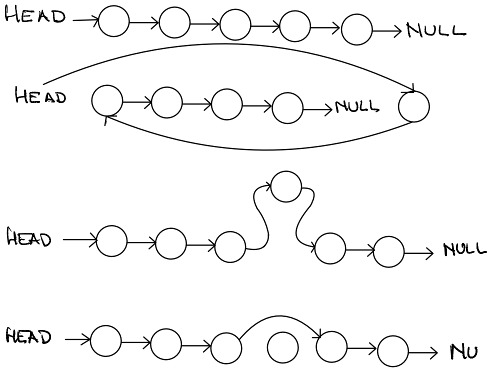
Running the program

```
int main(int argc, char* argv[])
{
  afunction();;
  return 0;
}

printing matrix using pointer to matrix: int(*pm)[dim/2] = m;
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
61 62 63
printing matrix using array of pointers to rows: int* r[dim]; int** pp = r;
11 12 13
21 22 23
31 32 33
41 42 43
51 52 53
61 62 63
```

Resizing

```
#include <cstring>
const double factor = 1.5;

int* resize(const int* array, const unsigned int size)
{
  const unsigned int newSize = size*factor;
  int* newP = new int[newSize];
  memcpy(newP, array, size*sizeof(int));
  return newP;
}
```

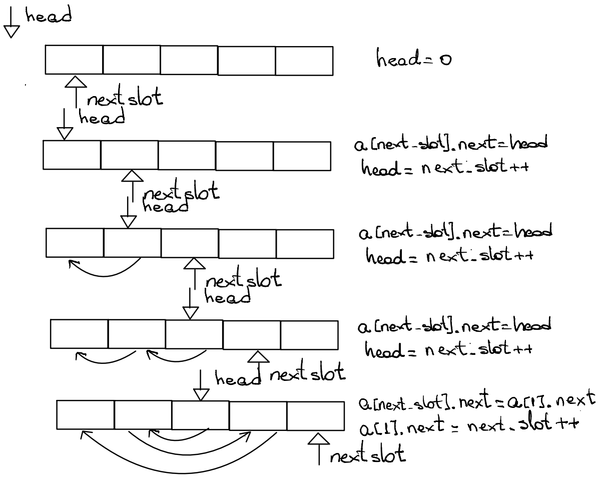
Linked List



Linked List

```
typedef unsigned int nodedata;
struct Node
{
  const nodedata     cargo;
  const unsigned int key;
  Node*              link;
  Node(const nodedata mycargo, const unsigned int akey) :
    cargo(mycargo), key(akey), link(NULL) { }
};
class LinkedList
{
public:
  LinkedList();
  ~LinkedList();
  unsigned int insert(nodedata cargo)
  { unsigned int nodekey = key; Node* newNode = new Node(cargo, nodekey);
    newNode->link = head; head = newNode; key++; count++; return nodekey; }
  unsigned int insertafterkey(nodedata, unsigned int key);
  void remove(unsigned int);
  nodedata removeafterkey(unsigned int key);
private:
  Node*        head;
  unsigned int count;
  unsigned int key;
};
```

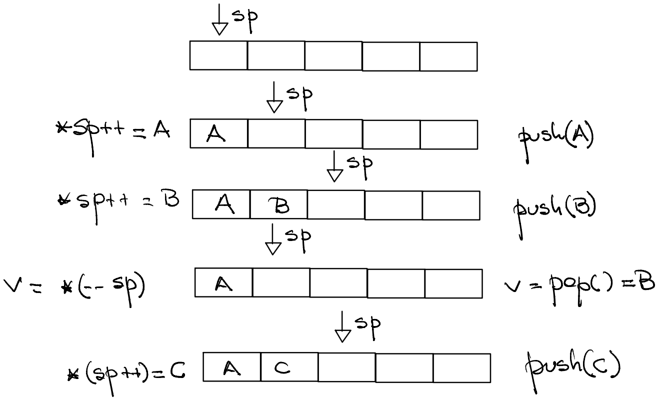
Array-based Linked List



Array-based Linked List Implementation

```
struct ArrayNode
{
  const nodedata     cargo;
  const unsigned int key;
  unsigned int       index;
  ArrayNode*         link;
  ArrayNode(const nodedata mycargo, const unsigned int akey) :
    cargo(mycargo), key(akey), index(0), link(NULL) { }
};
class ArrayLinkedList
{
public:
  ArrayLinkedList();
  ~ArrayLinkedList();
  unsigned int insert(nodedata);
  unsigned int insertafterkey(nodedata, unsigned int key);
  void remove(unsigned int);
  nodedata removeafterkey(unsigned int key);
private:
  ArrayNode*   array;
  unsigned int arraySize;
  unsigned int headIndex;
  unsigned int nextSlot;
  unsigned int emptySlotHead;
  unsigned int key;
};
```

Stack



Stack - Array Implementation

```
struct StackNode
{
  nodedata     cargo;
  StackNode() { };
};
class Stack
{
public:
  Stack(unsigned int maxdepth) : array(new StackNode[maxdepth])
  { sp=array; endStack = sp+maxdepth; }
  void push(const StackNode& n) { *sp++ = n; } // check sp < endStack
  StackNode pop() { return *(--sp); } // check sp>array
private:
  StackNode* sp;
  StackNode* array;
  StackNode* endStack;
};
```

Stack - Linked List Implementation

```
struct LLStackNode
{
  nodedata     cargo;
  LLStackNode* next;
  LLStackNode() { };
class LLStack
{
public:
  LLStack() : head(NULL) { }
  void push(const LLStackNode& n)
  { StackNode* sn = new LLStackNode(n); sn->next = head; head=sn; }
  // check head !=NULL
  LLStackNode pop() { LLStackNode sn = *(head); head=head->next; return sn; }
private:
  LLStackNode* head;
};
```

Example

```
a * (((b+c) * (d*e)) + f)
struct Item
{
  union
  {
  int var;
  char oper;
  } token;
  int tokenType;  // tokeType=0 for number, tokenType=1 for operator
};
     a * (((b+c) * (d*e)) + f)
     a (((b+c) * (d*e)) + f) *
     a (((b c +) (d e *) *) f +) *
Item* l = {a b c + d e * *  f + *}

     (a) b c + d e * *  f + *
push(a)       | a |

     a (b) c + d e * *  f + *
push(b)       | b | a |

     a b (c) + d e * *  f + *
push(c)       | c | b | a |
```

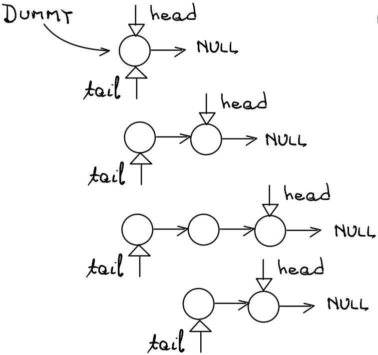
Example

```
     a b c (+) d e * *  f + *
c = pop()    | b | a |
b = pop()    | a |
h = b+c
push(h)      | h | a |

     a b c + (d) e * *  f + *
push(d)      | d | h | a |

     a b c + d (e) * *  f + *
push(e)      | e | d | h | a |

     a b c + d e (*) *  f + *
e = pop()    | d | h | a |
d = pop()    | h | a |
i = d*e
push(i)      | i | h | a |
```

Example

```
     a b c + d e * (*)  f + *
i = pop()    | h | a |
h = pop()    | a |
j = i*h
push(j)      | j | a |

     a b c + d e * *  (f) + *
push(f)      | f | j | a |

     a b c + d e * *  f (+) *
f=pop()      | j | a |
j=pop()      | a |
k = f+j
push(k)      | k | a |

     a b c + d e * *  f + (*)
k = pop()
a = pop()
result = k*a
```

Queues



head = tail = dummy

head→next = node
head = node

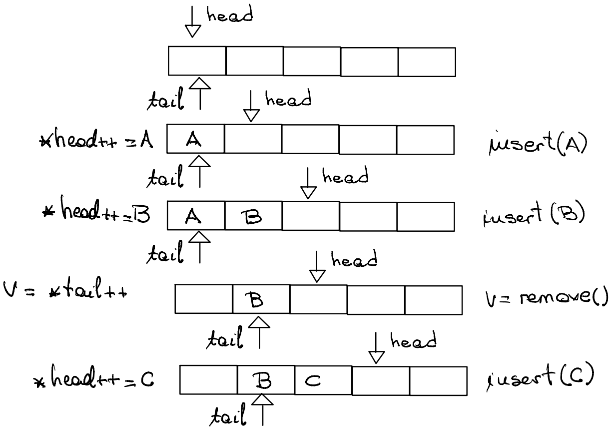
head→next = node
head = node

node = tail→next
delete tail
tail = node

Queue

```
struct QueueNode
{
  nodedata     cargo;
  QueueNode*   next;
  QueueNode() : next(NULL)  { };
};
class Queue
{
public:
  Queue() { QueueNode* dummy = new QueueNode; head=tail = dummy; };
  void insert(QueueNode& node)                       head->next=nn
  { QueueNode* nn = new QueueNode(node); nn->next = head->next; head=nn; }
  QueueNode remove() {
    QueueNode nn(*tail->next); QueueNode* tn = tail->next;
    delete tail; tail=tn; return nn; }
private:
  QueueNode* head;
  QueueNode* tail;
};
```

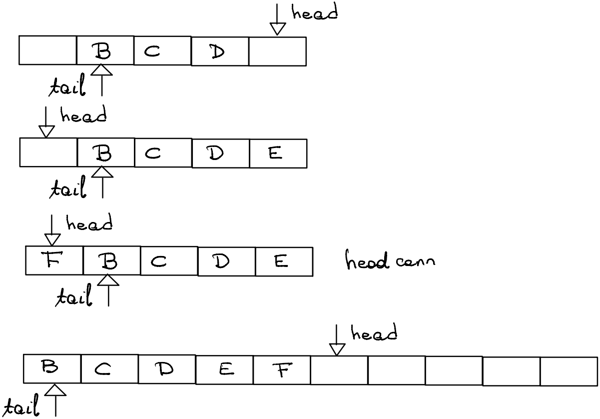
Array-based Implementation



Array-based Queue

```
struct ArrayQueueNode
{
  nodedata     cargo;
  unsigned int next;
  ArrayQueueNode() : next(0)  { };
};
class ArrayQueue
{
public:
  ArrayQueue(unsigned int maxSize) : maxQueueSize(maxSize)
  { head=tail= 0; array = new ArrayQueueNode[maxSize]; };
  void insert(ArrayQueueNode& node)
  { ArrayQueueNode nn = node;
    nn.next = head;
    array[head++] = nn; }
  ArrayQueueNode remove() {
    ArrayQueueNode nn = array[tail++]; return nn; }
private:
  unsigned int head;
  unsigned int tail;
  unsigned int maxQueueSize;
  ArrayQueueNode* array;
};
```

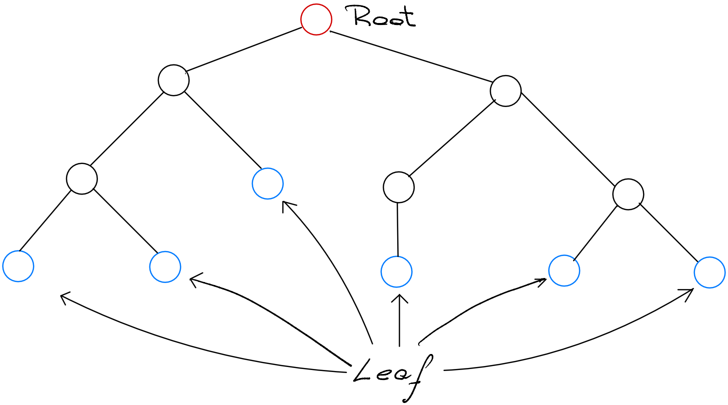
Resizing the Array



Wrap-around

```
class ArrayQueue2
{
public:
  ArrayQueue2(unsigned int maxSize) : maxQueueSize(maxSize)
  { head=tail= 0; array = new ArrayQueueNode[maxSize]; };
  void insert(ArrayQueueNode& node)
  { ArrayQueueNode nn = node;
    nn.next = head;
    array[head++] = nn;
    if (head == maxQueueSize ) head = 0;
    if (head == tail)
      { /* queue is full - raise exception
handle the execption by reallocating array and transferring items */
      }
  }
  ArrayQueueNode remove() {
    if (tail != head)
      { ArrayQueueNode nn = array[tail++]; return nn; }
    /* queue is empty - raise exception */ }
private:
  unsigned int head;
  unsigned int tail;
  unsigned int maxQueueSize;
  ArrayQueueNode* array;
};
```

Binary Trees



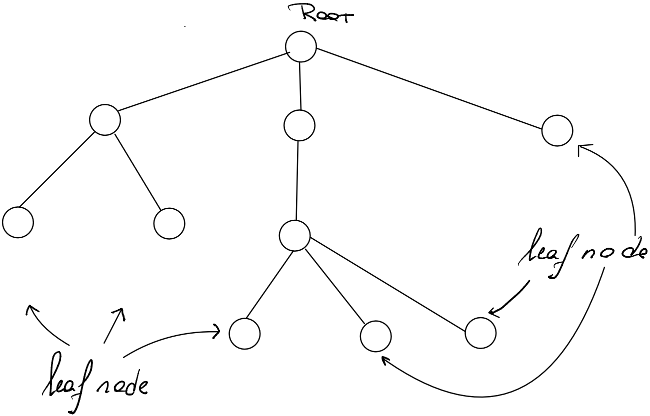
Properties of trees

- There is exactly one path connecting any two nodes in a tree
- A tree with *N* nodes hase *N-1* edges
- A binary tree with *N* internal nodes has *N+1* leaves
- The hight of a full binary tree with *N* internal nodes is about $log_2 N$

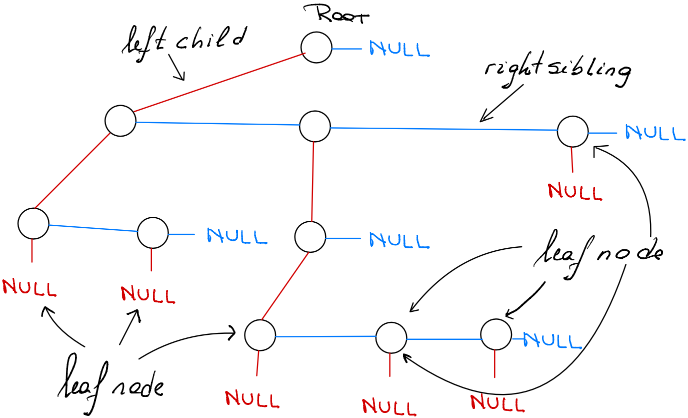
Binary Tree Representation

```
struct BinaryTreeNode
{
  BinaryTreeNode() : leftChild(NULL), rightChild(NULL) { }
  nodedata        cargo;
  BinaryTreeNode* leftChild;
  BinaryTreeNode* rightChild;
};

class BinaryTree
{
public:
  BinaryTree(BinaryTreeNode& root);
private:
  BinaryTree* rootNode;
};
```

General Tree Structure



General Tree Representation



Forest Representation

```
struct ForestNode
{
  ForestNode() : leftChild(NULL), rightChild(NULL) { }
  nodedata  cargo;
  ForestNode* leftChild;
  ForestNode* rightSibling;
};

class Forest
{
public:
  Forest(ForestNode& root);
private:
  ForestNode* rootNode;
};
```

Recursion

1. Function calls itself
2. *Termination* Condition

Mathematical recurrence $\iff$ Recursive programs
Factorial

$$N! = N \times (N-1)! \quad \text{for } N \geq 1 \qquad 0! = 1$$

```
int factorial(const int n)
{
  if (n==0) return 1;
  return n*factorial(n-1);
}
```

Recursion

Fibonacci Sequence

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } N \geq 2 \qquad F_0 = F_1 = 1$$

```
int fibonacci(const int n)
{
  if (n<2) return 1;
  return fibonacci(n-1)+fibonacci(n-2);
}
```

How many recursive calls are needed?
For F_0 or F_1: 1
For F_2: calls for F_0 + calls for F_1: 2
For F_3: calls for F_2 + calls for F_1: 3
For F_4: calls for F_3 + calls for F_2: 5
For F_N: ϕ^N where $\phi \approx 1.62$ - golden ratio

Divide and Conquer

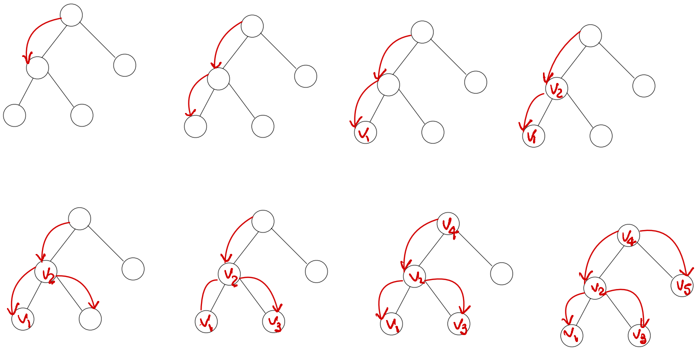
- Split input into 2 parts
- Two recursive calls, each operating on approximately half of the input
- Merge the results of processing 2 independent portions of the input
- Input is divided without overlap
- There may be code before, after, or in between ther recursive calls
- Top-down orientation

Tree Traversal

```
struct BinaryTreeNode
{
  BinaryTreeNode() : leftChild(NULL), rightChild(NULL) { }
  nodedata        cargo;
  BinaryTreeNode* leftChild;
  BinaryTreeNode* rightChild;
  void visit();
};

void inorder_traverse(BinaryTreeNode* t)
{
  inorder_traverse(t->leftChild);
  t->visit();
  inorder_traverse(t->rightChild);
}

void preorder_traverse(BinaryTreeNode* t)
{
  t->visit();
  preorder_traverse(t->leftChild);
  preorder_traverse(t->rightChild);
}
```

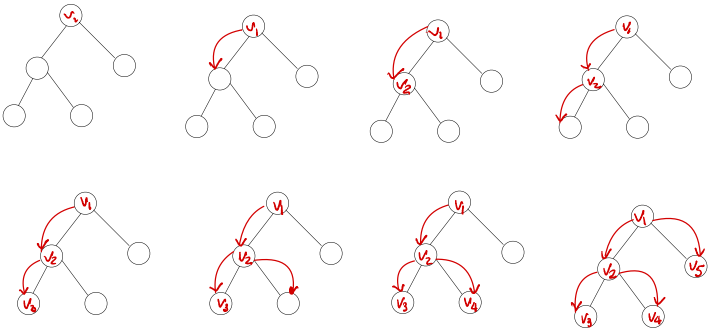
Inorder Traversal

traverse(t—>leftChild)
t—>visit()
traverse(t—>rightChild)



Preorder Traversal

```
t—>visit()
traverse(t—>leftChild)
traverse(t—>rightChild)
```



Removing Recursion

```
void preorder_traverse(BinaryTreeNode* t)
{
  if (t)
    {
    t->visit();
    preorder_traverse(t->leftChild);
    preorder_traverse(t->rightChild);
    }
}

void preorder_traverse(BinaryTreeNode* t)
{
    do
      {
      t->visit();
      if (t->leftChild) preorder_traverse(t->leftChild);
      t = t->rightChild;
      } while(t);
}
```

Removing Recursion

On entry

- push values of the local variables on the stack
- push the address of the next instruction (return) on the stack
- set the values of the parameters to the procedure
- go to the beginning of the procedure

On return

- pop the return address
- reset local variables
- go to the return address

Removing Recursion

```
void preorder_traverse(BinaryTreeNode* t)
{
  if (t)
    {
    t->visit();
    preorder_traverse(t->leftChild);
    preorder_traverse(t->rightChild);
    }
}
void preorder_traverse(BinaryTreeNode* t)
{
  do
    {
    while (t)
      {
      t->visit();
      stack.push(t);    // push the local variable into the stack
      t = t->leftChild; // set the new value for the function parameter
      }                 // returns to the begining of the while loop
// we are here when t->leftChild == 0
// meaning we have to return to the parent node
    if (stack.isempty()) break; // no more nodes on the stack. exit
    t = stack.pop();    // retrieve the pointer to the parent node
    t = t->rightChild;  // traverse the right subtree
    } while(1);         // reenter the function
}
```

Removing Recursion

```
void preorder_traverse(BinaryTreeNode* t)
{
  do
    {
    while (t)
      {
      t->visit();
      stack.push(t);
      t = t->leftChild;
      }
    if (stack.isempty()) break; --------------
    t = stack.pop(); ----------------------  |
    t = t->rightChild;                    |  |
    } while(1);                           |  |
}                                         |  |
void preorder_traverse(BinaryTreeNode* t) |  |
{                                         |  |
  stack.push(t);                          |  |
  while (!stack.isempty()) <------------------
    {                                     |
    t = stack.pop();  <--------------------
    if (t->rightChild) stack.push(t->rightChild);
    t->visit();
    if (t->leftChild) stack.push(t->leftChild);
    }
}
```

Algorithms and their Classification of Algorithms

- Constant Time
- $log\ N$: *logaritmic* running time
- N: *linear* running time
- $N\ log\ N$: *N log N* running time
- N^2: *quadratic* running time
- N^3: *cubic* running time
- 2^N: *exponential* running time

Computational Complexity - Big O notation

Definition: A function $g(N)$ is said to be $\mathrm{O}\big(f(N)\big)$ if there exist constants c_0 and N_0 such that $g(N)$ is less that $c_0 f(N)$ for all $N > N_0$