

Lecture #0

Tuesday, 20 June 2023 2:57 am

Standard template library which includes 3 types of generic entities: containers, iterators and algorithms. Algorithms can be applied to different data structures.

STL relieves programmers from writing their own implementations of various classes and functions instead they can use prepackaged generic implementations.

Containers

- ✧ Data structure that holds some objects that are usually of the same data type.
- ✧ Examples include deque, list, map, multimap, set, multiset, stack, queue, priority queue and vector.
- ✧ STL containers implemented as template classes include member functions specifying what the operations can be performed on the elements stored in the data structure specified by the container or the data structure itself.
- ✧ Member functions found by default are default and copy constructor, destructor, empty(), max_size(), size(), swap(), operator=, and all six relational operators [except in priority queue].

Iterators

- ✧ An object used to reference element stored in a container [a generalization of a pointer]
- ✧ They have the same dereferencing notation; similar arithmetic although all operators on iterators are not allowed in containers.

Algorithms

Vectors

- ✧ Container; a data structure with contiguous blocks of memory just like an array.
- ✧ Flexible array i.e. an array whose size can be dynamically changed.
- ✧ To use the vector class use #include <vector>

Lecture #1

Tuesday, 20 June 2023 2:28 am

Way of arranging data on computer efficiently. Divided in two categories:

- ▶ Linear data structure: elements are arranged in sequence-one after another.
 - Array data structure: data stored in continuous memory locations. All elements are of same data type.
 - Stack Data structure: LIFO-last in first out.
 - Queue: FIFO-first in first out.
 - Linked List: all elements are connected through a series of nodes, each node points to the next node(which contains data and another node address).
- ▶ Non-Linear data structure: no sequence. Instead arranged in hierarchical manner.
 - Graph data structure: each node is called vertex and each vertex is connected to other vertices through edges.
 - Spanning tree and minimum spanning tree
 - Strongly connected components
 - Adjacency matrix
 - Adjacency list
 - Tree data structure: collection of vertices, but there can only be one edge between two vertices.
 - Binary tree
 - Binary search tree
 - AVL tree
 - B- and B+ tree
 - Red-Black tree

LECTURE #3 (Theory)

Friday, 25 August 2023 4:03 pm

Templates

//printing array elements of integer array

```
Void print(int a[],int size)
{
    For(int i=0;i<size;i++)
    {
        Cout<<a[i]<<endl;
    }
}
```

//printing array elements of double array

```
Void print(double a[],int size)
{
    For(int i=0;i<size;i++)
    {
        Cout<<a[i]<<endl;
    }
}
```

- Here we have overloaded the print function as it has the same name however either it has different number of parameters or the datatypes of parameters is different or the order of parameters is different.
- This will allow the compiler to decide at compile time which function to call based on the parameters.
- The return type of the function doesn't matter, when overloading.
- If we want to make change to the print function we will have to change it in both functions which is time consuming therefore we look for something more generic.
- Abstraction will allow this program to work with many different data types. Also this allows reusability, writability and maintainability of the code.
- This generic programming can be done using TEMPLATES
 - Function Templates
 - Class Templates
- Syntax
 - template <typename T>
 - template <class X, class U>

// printing values of array using templates

```
Template <typename T>
Void print(T a[], int size)
{
    For(int i=0;i<size;i++)
    {
        Cout<<a[i]<<endl;
    }
}
```

```
Main()
{
    Char b[]={'a','b','c'};
    Int v[]={1,2,3};
    Int size=3;
    Print(v,size);
    Print(b,3);
}
```

- It is not necessary for function templates to have parameters. They can work without them too.

```
Template <typename T>
T getIn()
{
    T a;
    Cin>>a;
    Return a;
}
```

```

Main()
{
    int m;
    m=getIn();

    //the problem here is that in the function template any datatype can be given as input
    but we only require int input. Therefore, gives error
    // so we need to declare the datatype of the input we want EXPLICITLY

    m= getIn <int>();
}

```

- User-Defined Specialization
 - In case template cannot handle all datatypes.

INSERTION SORT

```

void insertionSort(int arr[], int size)
{
    int i=0,j=0;
    for(i=1; i<size ;i++)
    {
        int temp=arr[i];
        for(j=i ;j>0 && temp<arr[j-1];j--)
        {
            arr[j]=arr[j-1];
        }
        arr[j]=temp;
    }
}

```

BUBBLE SORT

```

void bubblesort(int arr[],int size)
{
    for(int i=0 ;i<size ;i++)
    {
        for(int j=0; j<size-1-i ;j++)
        {
            if(arr[j+1]<arr[j])
            {
                int temp=arr[j+1];
                arr[j+1]=arr[j];
                arr[j]=temp;
            }
        }
    }
}

```

```

void bubblesort(int arr[],int size)
{
    for(int i=0 ;i<size ;i++)
    {
        for(int j=i; j<size ;j++)
        {
            if(arr[i]>arr[j])
            {
                int temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            }
        }
    }
}

```

BINARY SEARCH

```

int binarySearch(int array[], int numelems, int value)
{
    int first = 0, last = numelems - 1, middle, position = -1;
    bool found = false;
    while (!found && first <= last)
    {

```

```

middle = (first + last) / 2; // Calculate mid point
if (array[middle] == value)
{ // If value is found at mid
    found = true;
    position = middle;
}
else if (array[middle] > value) //If value is in lower half
    last = middle - 1;
else
    first = middle + 1;
// If value is in upper half
}

return position;
}

```

SELECTION SORT

```

void selectionsort(int arr[],int size)
{
    int rightmost=size-1;
    int max_index=0;

    for(rightmost ;rightmost>=0 ;rightmost--)
    {
        max_index=0;
        for(int i=1; i<rightmost;i++)
        {
            if(arr[max_index]<arr[i])
                max_index=i;
            if(arr[rightmost]<arr[max_index])
            {
                int temp=arr[rightmost];
                arr[rightmost]=arr[max_index];
                arr[max_index]=temp;
            }
        }
    }
}

```

QUICKSORT

```

#include <iostream>
using namespace std;
int divide(int arr[], int low, int high)
{
    int i = low - 1;
    float c = arr[high]; //this is the last index of the array (value of salary )

    for (int j = low; j <= high-1; j++)
    {
        if (c >= arr[j]) //checks if the slary at position j is less than the max index salary
        {
            i++;
            int temp = arr[i]; //swapping
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    float temp = arr[i + 1]; //swapping current i index with the max array
    arr[i + 1] = arr[high];
    arr[high] = temp;
    i += 1;
    return i ;
}

void iterativeQuickSortEmployees(int arr[], int size)
{
    int* a_stack = new int[size];
    int top = -1;

    a_stack[++top] = 0;
    a_stack[++top] = size - 1;

    while (top >= 0)
    {
        int high = a_stack[top--];

```

```

    int low = a_stack[top--];

    int num = divide(arr, low, high);    //pivot index

    if (low<num-1)
    {
        a_stack[++top] = low;
        a_stack[++top] = num - 1;
    }

    if (high>num + 1 )
    {
        a_stack[++top] = num + 1;
        a_stack[++top] = high;
    }
}

int main()
{
    int p[]={10,20,11,3,1,5,2};
    int size = sizeof(p)/sizeof(p[0]);
    iterativeQuickSortEmployees(p, size);
}

```

TimeComplexity

Wednesday, 6 September 2023 7:39 pm

- ◇ An algorithm is a finite sequence of instructions which has a clear meaning and can be performed with a finite amount of effort in a finite time.
- ◇ Performance of a program can be judged from space usage and time usage
- ◇ Time complexity is the amount of time a program needs to run to completion
- ◇ Amount of memory needed to run to completion is called space complexity.
 - Instruction space: the amount of space used by instructions which depends on:
 - The compilers used to complete the program into machine code
 - The target computer
 - The compiler options in effect at the time of compilation
- ◇ Algorithm Goals
 - Save time
 - Save space
 - Save face
- ◇ Clarification of Algorithms
 - If n is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc
 - 1 : instructions executed once or at most only a few times. If all the instructions of the program have this property we say that its running time is a constant.
 - Log n : when the running time of a program is logarithmic, the program gets slightly slower as n grows. Whenever n increases log n increases by a constant
 - n when the running time of a program is linear.
 - $n \cdot \log n$: when solving a problem by breaking it up into smaller sub-problems, solving them independently and then combining the solutions.
 - n^2 : running time is quadratic. Practical for relatively smaller program. Eg when double loops are used
 - n^3 : practical for small problems
 - 2^n : when n doubles the running time squares
- ◇ Complexity of algorithm
 - Best case: the minimum possible value of $f(n)$
 - Worst case: the maximum value of $f(n)$ for any key possible input
 - Average case: the expected value of $f(n)$
- ◇ Efficiency of algorithms= Analysis of algorithms

3.3: Logarithmic time — $O(\log n)$.

```
1 def logarithmic(n):
2     result = 0
3     while n > 1:
4         n //= 2
5         result += 1
6     return result
```

The value of n is halved on each iteration of the loop. If $n = 2^x$ then $\log n = x$. How long would the program below take to execute, depending on the input data?

3.4: Linear time — $O(n)$.

```
1 def linear(n, A):
2     for i in xrange(n):
3         if A[i] == 0:
4             return 0
5     return 1
```

Let's note that if the first value of array A is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array A does not contain any 0.

3.5: Quadratic time — $O(n^2)$.

```
1 def quadratic(n):
2     result = 0
3     for i in xrange(n):
4         for j in xrange(i, n):
5             result += 1
6     return result
```

The result of the function equals $\frac{1}{2} \cdot (n \cdot (n + 1)) = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$ (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms ($\frac{1}{2} \cdot n$ in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

3.6: Linear time — $O(n + m)$.

```
1 def linear2(n, m):
2     result = 0
3     for i in xrange(n):
4         result += i
5     for j in xrange(m):
6         result += j
7     return result
```

3.3: Logarithmic

```
1 def logarithmic(n):
2     result = 0
3     while n > 1:
4         n //= 2
5         result += 1
6     return result
```

The value of n is halved
would the program be

3.4: Linear time

```
1 def linear(n, A):
2     for i in xrange(n):
3         if A[i] == 0:
4             return 0
5     return 1
```

Let's note that if the first
remember, when analyzing
will take the longest ti

Linked List/Stack/Queue

Sunday, 24 September 2023

11:10 am

Operations linked list

- Insert
- Delete
- Sort
- Reverse
- Search
- Replace
- Remove duplicates

QUEUE

{may be used in BFS (breadth first search and tree transversal)}

- ❖ Inserted at rear
- ❖ Removed from front
- ❖ When inserting increment rear by 1
- ❖ When deleting decrement by 1
- ❖ When only one value in the queue the rear == front
- ❖ When empty front == rear == -1
- ❖ Three types of implementation
 - Array based
 - Linked list based
 - Pointer based implementation

❖ **ARRAY BASED**

- Has two counter front and rear
- Both are set to -1 when queue is empty
- Increment rear by 1 when inserting and decrement front by 1 when removing

front



□ PROBLEM

rear



○

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

- Method 1: create circular queue
 Eg: if (rear == queuesize-1)
 Rear = 0;
 Else
 Rear++;
 OR
 Rear = (rear+1)%queuesize
- PROBLEM: how to avoid overwriting an element
 - METHOD 1: create a variable of num of elements
 - If num_elements == queuesize-1 then
 QUEUE is FULL

❖ LINKED LIST BASED

- The queue has a NODE pointer of front and rear
- The NODE itself has two variables such as
 data(int) and next(NODE*)

❖ Operations on queue

- Enqueue
- Dequeue
- Reverse
- Sort
- Remove duplicates
- Print
- Clear
- Is_empty/is_full

Stack

- used for string reversal
- Expression evaluation and conversion
- Function call
- Parentheses checking

← top

- LIFO or FILO
- Insertions and deletion happens at top

- Static implementation using array
- Dynamic used linked lists

3	0
2	1
1	2
..(value)	Index

- Here values are added at the 0th index problem is each time the rest values must be move one down therefore not efficient.
- And for removing whole array must be moved up
- Another way is letting the stack grow downwards
- And the stack is full ~~when the top == stacksize-1~~ ← **top**

1	0
2	1
3	2
..(value)	..(index)

→ → Null

- Pointer-based/ Linked list based
- Has a node which has data and next

1		2		3		4	
---	--	---	--	---	--	---	--

Monday, 23 October 2023 7:55 pm

as
af node
/2
leaf

Leaf nodes= $(n+1)/2$



A complete binary tree with n nodes has height $\log_2(n+1)-1$

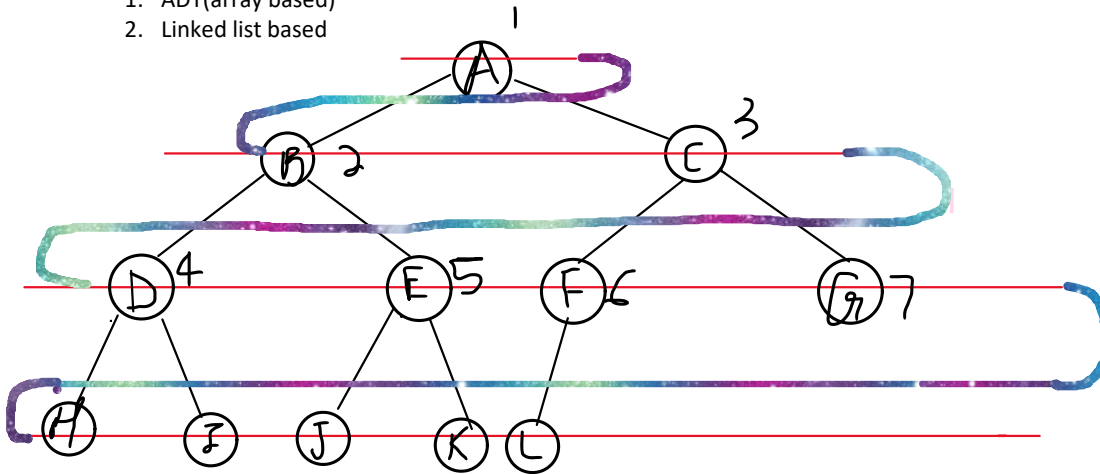
- left and right sub-trees if every node has the same height

Binary tree Implementation

Monday, 23 October 2023 8:25 pm

Two methods of implementation:

1. ADT(array based)
2. Linked list based



1	2	3	4	5	6	7	8	9	10	11	12			
A	B	C	D	E	F	G	H	I	J	K	L	(empty)	(empty)	(empty)

Right node of A

$$2k+1 === 2(1)+1=3$$

Left node of A

$$2k === 2(1)=2$$

Right node of E

$$2k+1 === 2(5)+1=11$$

Left Node of E

$$2k === 2(5)=10$$

Parent Node

$$5/2=2$$

Degree of Node	Degree of a node implies the number of child nodes a node has
Internal Node	A node that has at least one child is known as an internal node.
Ancestor node	An ancestor or ancestors to a node are all the predecessor nodes from root until that node. I.e. any parent or grandparent and so on of a specific node are its ancestors.
Descendant	Immediate successor of a node is its descendent.

Number of edges: If there are 'n' nodes in a tree then there would be $n-1$ edges. Each edge is the line-arrow connecting two nodes.

Depth of node x: Depth of a specific node x is defined as the length from root till this x node. One edge contributes to one unit in the length. Hence depth of a node x can also be considered as the number of nodes from root node till this x node.

Sorting Algo

Saturday, 4 November 2023

4:23 pm

1. **The Time Complexity of Insertion Sort:** The time complexity of Insertion Sort is $\Omega(n)$ in its best case possible and $O(n^2)$ in its worst case possible. It has been observed that for very small 'n', the Insertion Sort is faster than more efficient algorithms such as Quick sort or Merge Sort.
2. **The Time Complexity of Quick Sort:** The time complexity of Quick Sort is $\Omega(n \log n)$ in its best case possible and $O(n^2)$ in its worst case possible. As we know [QuickSort](#) is a Divide and Conquer algorithm where the element is picked as a pivot and partitions are made on the given array around the picked pivot. This makes it the fastest of the sorting algorithms as due to its performance in best case and also in average cases are said to be of $O(n \log n)$.
3. **The Time Complexity of Bubble Sort:** The time complexity of Bubble Sort is $\Omega(n)$ in its best case possible and $O(n^2)$ in its worst case possible. As is widely known that the Time Complexity of Bubble Sort is a reliable sorting algorithm as it runs through the list repeatedly, compares adjacent elements, and swaps them if they are out of order. This process is repeated until the list is sorted and hence, this is also considered the simplest sorting algorithm.

QUICK SORT

```
// finding the partition point & rearranging the array
int partition(int array[], int low, int high)
{
    // selecting the rightmost element as pivot
    int pivot = array[high];
    // setting the left pointer to point at the lowest index initially
    int left = low;
    // setting the right pointer to point at the highest index initially
    int right = high - 1;
    // running a loop till left is smaller than right
    while(left <= right)
    {
        // incrementing the value of left until the value at left'th index is smaller than pivot
        while(array[left] < pivot){
            left++;
        }

        // decrementing the value of right until the value at right'th index is greater than pivot
        while(array[right] > pivot){
            right--;
        }

        if(left < right){
            // swapping the elements at left & right index
            swap(array[left], array[right]);
        }
    }
    // swapping pivot with the element where left and right meet
    swap(array[left], array[high]);
    // return the partition point
    return (left);
}
```

```
void quickSort(int array[], int low, int high) {  
    if (low < high) {  
  
        // since this function returns the point where the array is partitioned, it is used to track the  
        subarrays/partitions in the array  
        int pi = partition(array, low, high);  
  
        // recursively calling the function on left subarray  
        quickSort(array, low, pi - 1);  
  
        // recursively calling the function on right subarray  
        quickSort(array, pi + 1, high);  
    }  
}
```

(BST) Binary Search Tree

Saturday, 4 November 2023

4:57 pm

- ❖ If you want to insert duplicates in BST then add a variable count in the Node (assuming node based implementation) this will keep track of how many time a value is repeated

❖ NODE BASED IMPLEMENTATION

```
Node
{
    Int data;
    Node* left;
    Node* right;
}
```

◆ INSERT

- If root== nullptr
 - Insert new node here to root and return
- Else if (val < root->data)
 - Check if left is empty [root->left]
 - Else
 - Move to cur=cur->left
 - Else check if (root->data<data)
 - Move to right check if root->right is empty
 - Else move to cur=cur->right

❖ SEARCH

- If root==null return 0
- If val<root->data go to left subtree ..repeat from step 1
- If val>root->data go to right subtree... repeat from step 1
- If(val==cur->data) return 1

❖ PREORDER TRANSVERSAL

- Root->left->right

❖ INORDER TRANSVERSAL

- Left->Root->right

❖ POSTORDER TRANSVERSAL

- Left->right->Root

- ❖ Predecessor of a node

- Predecessors can be described as the node that would come right before the node you are currently at. To find the predecessor of the current node, look at the right-most/largest leaf node in the left subtree.

- ❖ Successor of a node

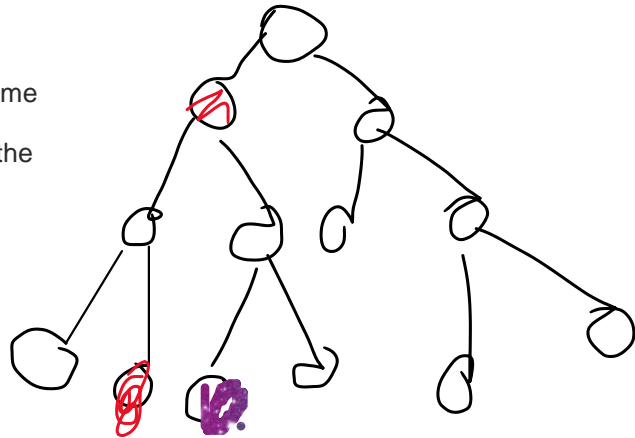
- Successors can be described as the node that would come right after the current node. To find the successor of the current node, look at the left-most/smallest leaf node in the right subtree.

- ❖ Height of tree

- o If (root==nullptr)
 - Return 0;
- o Return 1+ height(root->left)+height(root->right)

❖ MAXDEPTH

```
int maxDepth(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        int rDepth = maxDepth(node->right);
        int lDepth = maxDepth(node->left);
        if (lDepth > rDepth)
        {
            return(lDepth+1);
        }
        else
        {
            return(rDepth+1);
        }
    }
}
```



Hashing

Monday, 20 November 2023 8:19 pm

- For storing and retrieving data from database in constant time
- Mapping technique
- Larger values mapped onto larger
- Hash table is a database that allows storage of data
- Using hash function we don't need to scan like in array and are able to store in constant time

Hash Function Methods

- $K \bmod 10$
 - ❖ Here k is the unique value of database
- $K \bmod n$
- Division method
- Multiplication method
- Mid square
 - ❖ Ex when value is 123 then store at 2^2 ie 4
- folding method
 - ❖ Suppose data is 123456
 - ❖ Suppose table is between 0-999
 - ❖ So fold and add $123+456$ and store answer in table
- At times when using any of the above methods the index generated might already have a value this would lead to COLLISION

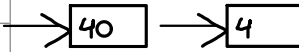
How to deal with collision (techniques of collision resolution)

- Chaining/ open hashing
 - ❖ Insert == $O(1)$; Search == $O(n)$; Delete == $O(n)$
 - ❖ Make external chains
 - ❖ Insertion of linked list happens at head INSERTATHEAD
 - ❖ In case there is already a value at the index then create a linked list and chain to that index

0	12
1	23
2	3
3	11

- ❖ Now suppose you have to insert 40 at index 1
- Open addressing/closed hashing
 - ❖ Utilizing the space initially provided

0	12
1	23
2	
3	11



- ❖ If value 40 is to be inserted at index 1 then as it is already filled so in case of LINEAR PROBING just check the next index in the given space till last index is reached
 - ❖ In this case 40 is inserted at index 2
 - ❖ Insert == ; Search == $O(n)$; Deletion
 - ❖ Index = index % hashtable size
 - ❖ Index = (index + 1) % hashtable size
 - ❖ Index = (index + 2) % hashtable size ...
 - ❖ In case of QUADRATIC PROBING $h + i^2 \bmod n$ here h is the hash value.
 - ❖ Suppose hash value is 0 then $0 + 1^2$ if this index is also filled then $0 + 2^2$ and so on
 - ❖ Double hashing is when two hash functions are used one is for inserting and other is for inserting in case of collision
- Rules of Hashing
 - ❖ The hash value is fully determined by the data being hashed
 - ❖ The hash function uses all input data
 - ❖ The hash function uniformly distributes the data across the entire set of possible hash values
 - ❖ The hash function generates very different hash value for similar things

→ Methods to create hash function (Open hashing)

- ❖ Multiplication method
 - Multiply the key by a constant A that is between 0 and 1

- Extract the fractional part of kA
- Multiply the fractional part by m (which is the hash table size)
- Eg
 - If $\text{key}=6$ $A=0.3$ $m=32$
 - $K \times A = 6 \times 0.3 = 1.8$
 - Extract fractional part ie 0.8
 - $M \times 0.8$ ie $32 \times 0.8 = 25.6$
 - Thus $\Rightarrow h(6) = 25$

❖ Division method

- Key must be transformed into an integer value
- The value must be telescoped into range 0 to $m-1$
- This method includes the above mentioned modulus methods
- However problem with this is the certain values of m are bad because they can lead to collision
 - Power of 2
 - Non-prime numbers
 - Power of 10
- It is best to make value of hash function dependent of all values of the key

- Linear probing can lead to the formation of clusters in case they all have the same hash value then they will be placed in continuous memory
- This is called as primary clustering
- They further merge with other clusters and form bigger clusters
- Quadratic probing leads to the formation of mild cluster or secondary clustering
- Clustering can be improved by moving the hash function to cubic but this increases the cost

Primary clustering the increase in probability of element to be stored in a particular index ; increase search time as the whole cluster must be searched for the value

Secondary clustering when two or more elements are competing for same probe sequence

Comparison of Open Addressing Techniques-

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Load Factor (α)

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

In open addressing, the value of load factor always lie between 0 and 1.

B-TREES

Saturday, 25 November 2023 7:36 pm

Disk structure

- ⊙ The disk is a circular structure. Divided in tracks and sectors.
- ⊙ Any location on disk can be accessed by track number and sector number
- ⊙ Typically block size is 512 bytes
- ⊙ A head writes/reads data from the disk .
- ⊙ For accessing data from HDD RAM is used

How is data stored on disk

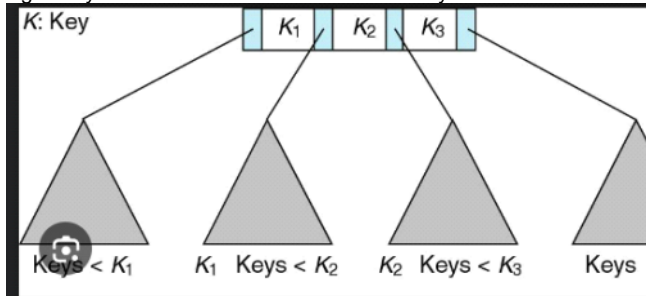
⊙

What is indexing

What is multilevel indexing

M-way search tree

- ⊙ Each tree can have m children and m-1 keys/nodes
- ⊙ Eg 4-way search tree has 4 children and 3 key/node



- ⊙ B-tree
 - ⊙ All leaf nodes are at the same level
 - ⊙ All non-leaf nodes except root have at most m and at least m/2 children
 - ⊙ The number of keys is one less than the number of children i.e. for non-lead nodes we have at most m-1 nodes and at least m/2 leaf nodes
 - ⊙ The root may have as few as 2 children unless it is the root node only
- ⊙ Insert
 - ⊙ First find the leaf node to which X should be added
 - ⊙ Check the value in the nodes after adding the key if value is fewer then $2t-1$ than no changes required else if equal to $2t$ then overflowed
 - ⊙ To fix the overflowed split the node into 3 parts
 - ⊙ Left the first t values become the left child
 - ⊙ The value at middle becomes the parent of new left and right node
 - ⊙ Right the last t-1 values become the right child node
 - ⊙ Complexity $O(h)=O(\log, n)$

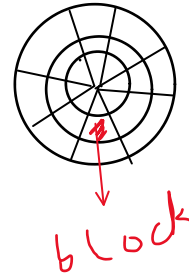
Insertion Algorithm

1. Insert the key in its leaf in sorted order
2. If the leaf ends up with $L+1$ items, **overflow!**
 - Split the leaf into two nodes:
 - original with $\lceil (L+1)/2 \rceil$ items
 - new one with $\lfloor (L+1)/2 \rfloor$ items
 - Add the new child to the parent
 - If the parent ends up with $M+1$ children, **overflow!**
3. If an internal node ends up with $M+1$ children, **overflow!**
 - Split the node into two nodes:
 - original with $\lceil (M+1)/2 \rceil$ children
 - new one with $\lfloor (M+1)/2 \rfloor$ children
 - Add the new child to the parent
 - If the parent ends up with $M+1$ items, **overflow!**
4. Split an overflowed root in two and hang the new nodes under a new root
5. Propagate keys up tree.

This makes the tree deeper!

21

- ⊙ Delete



Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow!**

- Adopt data from a neighbor; update the parent
- If adopting won't work, delete node and merge with neighbor
- If the parent ends up with fewer than $\lceil M/2 \rceil$ children, **underflow!**

29

Deletion Slide Two

3. If an internal node ends up with fewer than $\lceil M/2 \rceil$ children, **underflow!**

- Adopt from a neighbor; update the parent
- If adoption won't work, merge with neighbor
- If the parent ends up with fewer than $\lceil M/2 \rceil$ children, **underflow!**

4. If the root ends up with only one child, make the child the new root of the tree

This reduces the height of the tree!

5. Propagate keys up through tree.

30

- Used for disk access
- A tree of order m can have max m children and minimum $\text{ceil}(m/2)$ children
- A tree of order m can have max $m-1$ keys and minimum $\text{ceil}(m/2)-1$ keys
- This doesn't apply to root node a root node must have at least two children
- Follows the properties of BST when inserting

Sr. No.	Algorithm	Time Complexity
1.	Search	$O(\log n)$
2.	Insert	$O(\log n)$
3.	Delete	$O(\log n)$

- The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:

$$h_{min} = \lceil \log_m(n + 1) \rceil - 1$$

- The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is:

$$h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor$$

and

$$t = \lceil \frac{m}{2} \rceil$$

	Root node	Non-root node
Minimum number of keys	1	$\lceil m/2 \rceil - 1$
Minimum number of non-empty subtrees <i>children</i>	2	$\lceil m/2 \rceil$
Maximum number of keys	$m - 1$	$m - 1$
Maximum number of non-empty subtrees <i>children</i>	m	m

Binary Heap

Sunday, 26 November 2023

5:06 pm

- ⊙ Either a MinHeap or MaxHeap
- ⊙ In a minheap the value at the root must be minimum among all the nodes present in heap
- ⊙ A binary heap is a (almost) complete binary tree
- ⊙ The bottom level must be partially filled from left to right

GRAPHS

Saturday, 16 December 2023 3:12 pm

- A **path** is a sequence of nodes in the order we are visiting. No nodes are repeated in a path except when the source and destination are same
 - **Walk** is a sequences of vertices and edges of a graph. In a walk vertices and edges can be repeated
 - If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed **simple path**.
 - **Open walks**: when initial and final vertices are different
 - **Closed walk**: when the initial and final vertices are same
 - **Trail**: an open walk with no edge if repeated; vertex can be repeated
 - **Circuit**: a graph transversal tht is closed in which there is no edge repeated but vertex can be repeated
 - **Cyclic graph**: which starts and ends at the same vertex
 - A **cycle** can be defined as the path which has no repeated edges or vertices except the first and last vertices.
 - A **connected graph** is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph
 - A **complete graph** is the one in which every node is connected with all other nodes. A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.
-
- Adjacency List and Adjacency Matrix
 - Adjacency List
 - When there are more edges are less than the nodes
 - Adjacency Matrix
 - Dense graph when more edges than nodes
 - DFS
 - Use stack
 - Used to find cycles in directed and undirected graphs
 - BFS
 - Use queue
 - Finding shortest path in undirected graph
 - For the root node push it in queue
 - In a loop (works till its not empty)
 - There can be a maximum n^{n-2} number of spanning trees that can be created from a complete graph.
 - A spanning tree has **n-1** edges, where 'n' is the number of nodes.