|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C:\Documents and Settings\Litao\Desktop\12.jpgC:\Documents and Settings\Litao\Desktop\fdsf.jpg**Asymptotic Notations – Big Oh(0)**  The O (pronounced big-oh) is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.C:\Documents and Settings\Litao\Desktop\234.jpg    The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. Measuring efficiency of algorithm based on time taking input as size given to function  **Big 0 Example:**  Example 2[quadric function] f(n) = 10n^2+4n+2  For n >= 2, f(n) <= 10n^2+5n.  For n>= 5, 5n < n^2.  Hence for n>= n0 = 5, f(n) <= 10n^2+n^2 = 11n^2. Therefore, f(n) = O(n^2).  The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Theta notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases: 1. The worst case time complexity of Insertion Sort is Theta (n^2). 2. The best case time complexity of Insertion Sort is  Theta(n). The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm(No of times a single input is touched).  O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 <= f(n) <= cg(n) for all n >= n0}  **Omega Example :**  f(n) = 3n+3 > 3n for all n. So f(n) = Omega(n)  Just as Big O notation provides an asymptotic upper bound on a function,  Omega notation provides an asymptotic lower bound.   Notation< can be useful when we have lower bound on time complexity of an algorithm. For a given function g(n), we denote by  Omega(g(n)) the set of functions.  Omega (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= cg(n) <= f(n) for all n >= n0}.  **Theta Example :**  f(n) = 3n+3 is Theta(n), since n <= 3n+3 <= 4n, when n >= 3.  Similarly, f(n) = 3n+2 is Theta(n)  f(n) = 5n^2 - 10n + 9 is Theta(n^2)  The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.3n3 + 6n 2 + 6000 = Theta (n3) Dropping lower order terms is always fine because there will always be a n0 after which  (n3) beats   (n2) irrespective of the constants involved.For a given function g(n), we denote  (g(n)) is following set of functions.((g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1\*g(n) <= f(n) <= c2\*g(n) for all n >= n0}. The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1\*g(n) and c2\*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0  **Amortized Analysis:**Amortized analysis refers to determining the time-averaged running time for a sequence of operations.The amortized cost of n operations is the total cost of the operations divided by n.The motivation for amortized analysis is that looking at the worst-case time per operation can be too pessimistic if the only way to produce an expensive operation is to “set it up” with a large number of cheap operations beforehand.**eg:** Say we want to use an array to implement a stack. We have an array A, with a variable top that points to the top of the stack (so A[top] is the next free cell). This is pretty easy: To implement push(x), we just need to perform: A[top] = x; top++; To implement x=pop(), we just need to perform: top--; x = A[top]; (first checking to see if top==0 of course...) However, what if the array is full and we need to push a new element on? In that case we can allocate a new larger array, copy the old one over, and then go on from there. This is going to be an expensive operation, so a push that requires us to do this is going to cost a lot. But maybe we can “amortize” the cost over the previous cheap operations that got us to this point. So, on average over the sequence of operations, we’re not paying too much. To be specific, let us define the following cost model.  Cost model: Say that inserting into the array costs 1, taking an element out of the array costs 1,  and the cost of resizing the array is the number of elements moved. (Say that all other operations, like incrementing or decrementing “top”, are free.) Question 1: What if when we resize we just increase the size by 1. Is that a good idea? Answer 1: Not really. If our n operations consist of n pushes then even just considering the array-resizing cost we will incur a total cost of at least 1 + 2 + 3 + 4 + . . . + (n − 1) = n(n − 1)/2. That’s an amortized cost of (n − 1)/2 per operation just in resizing. Question 2: What if instead we decide to double the size of the array when we resizeThis is much better. Now, in any sequence of n operations, the total cost for resizing is 1 + 2 + 4 + 8 + . . . + 2i for some 2i < n (if all operations are pushes then 2i will be the largest power of 2 less than n). This sum is at most 2n − 1. Adding in the additional cost of n for inserting/removing, we get a total cost < 3n, and so our amortized cost per operation is < 3.  Not just consider one operation, but a sequence of operations on a given data structure. Average cost over a sequence of operations.  Probabilistic analysis: Average case running time: average over all possible inputs for one algorithm (operation).  If using probability, called expected running time.  Amortized analysis: No involvement of probability  Average performance on a sequence of operations, even some operation is expensive. Guarantee average performance of each operation among the sequence in worst case.  **List Size Example :**  if (arrSize >= capacity) {  capacity = capacity \* 2;  KeyValue \*\*newarrayValues = new KeyValue\*[capacity];  for (int i = 0; i < arrSize; i++) {  newarrayValues[i] = arrayValues[i]; }  delete[] arrayValues;  arrayValues = newarrayValues; }  arrayValues[arrSize] = node;  arrSize++; | **Merge Sort : nlogn, Not in Place, In place**  Merge Sort - O(nlogn) - Best Case, worst case and average case In computer science, merge sort (also commonly spelled mergesort) is an O(n log n) comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output  Divide the array into two, take the first part, divide until its1  then join the split one, join the outer one, then once the left array is sorted, take the second array and then again perform the divide and finally join the big ones  void mergeSort(int \*arr,int n) {  int \*left;int \*right;if (n < 2) return;  int mid = n / 2;left = (int\*)malloc(mid\*sizeof(int));  right = (int\*)malloc((n - mid)\*sizeof(int));  for (int i = 0; i < mid; i++) {  left[i] = arr[i];}  for (int i = mid; i < n; i++) {  right[i-mid] = arr[i]; }  mergeSort(left, mid);  mergeSort(right, n - mid);merge(arr,n,left,mid,right,n-mid);}  void merge(int \*arr,int n,int \*left,int leftCount,int\* right,int rightCount) {  int i =0, j = 0, k = 0;  while (i < leftCount && j < rightCount) {  if (left[i] < right[j]) arr[k++] = left[i++];  else arr[k++] = right[j++];}  while (i < leftCount) { arr[k++] = left[i++];}  while (j < rightCount) {arr[k++] = right[j++];}}  int main() {  int n; std::cout << "Enter the Size of the list";  std::cin >> n; int \*a = new int[n];  std::cout << "Enter the element";  for (int i = 0; i < n; i++) {  std::cin >> a[i]; }  //mergeSort(a,n);  //int numberOfElements = sizeof(a) / sizeof(a[0]);  //std::cout << "Number of Elements :" << numberOfElements;  std::cout << "\n After Sorting \n";  for (int i = 0; i < n; i++) { std::cout << " " << a[i];}}  **Selection Sort**  In Memory - Doesnt need Auxilary array , Not stable  In selection sort, we consider the first element to be min, then start comparing the first element with all the remanining elements, if there are any small elements we swap the first and the min, So at the end of the first pass, the smallest element will be in the last position.  void selection\_sort(int \*a,int n) {  int temp; for (int i = 0; i < n; i++) {  int small = i; for (int j = i + 1; j < n; j++) {  if (a[small]>a[j]) {  small = j;} } if (small > i) { temp = a[i];  a[i] = a[small]; a[small] = temp; } } }  Best Case, worst case and average case -> O(n2)  It needs to compare all the time with other elements  **Bucket Sort**  Set up an array of initially empty buckets.  Go over the original array, putting each object in its bucket.  Sort each non-empty bucket.  Visit the buckets in order and put all elements back into the original array  Value/Max Number \* number of buckets;  Add the element to the bucket. Using a linked list this is O(1)  Going througt the list and put the ellements in the correct bucket = O(n)  Merging the buckets = O(k)  O(1)\*O(n)+O(k) = O(n+k)  worst case -> O(n^2)  Worst case performance : O(n^2)  Best case performance : Omega(n+k)  Average case performance : Theta(n+k)  **Bubble Sort - In Memory - Doesnt need auxilary array**  In Bubble sort we compare adjecent elements one by one, After first pass the highest element will be in last position.  In the second pass, we will run the array from first to last -1,since last one is already sorted In third pass, It will be first to last - 2  Worst case is when it is in Reverse Order, need to sort everything  Best case is when it is already sorted.  Worst case O(n2), best case O(n) and average its (nlogn)  Best case will be n because there will no swaps involved and hence only comparison. So its n \*1 = n;  **Insertion Sort** : Stable, In place  Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.  Best Case : O(n) When the array is sorted and there is only comparisons and there is no swap required  Worst Case : O(n^2) when the array is not sorted and there are n swaps  3 7 4 9 5 2 6 1- 3 7 4 9 5 2 6 1 - 3 7 4 9 5 2 6 1- 3 4 7 9 5 2 6 1  3 4 7 9 5 2 6 1-3 4 5 7 9 2 6 1- 2 3 4 5 7 9 6 1-2 3 4 5 6 7 9 1-1 2 3 4 5 6 7 9  void insertion\_sort(int \*arr, int length){  int j, temp; for (int i = 1; i < length; i++){j = i;  while (j > 0 && arr[j] < arr[j - 1]){  temp = arr[j]; arr[j] = arr[j - 1];  arr[j - 1] = temp; j--; } } } | **HeapSort : Inplace, nlogn, not stable**  **Max Heap:**  The MAX-HEAPIFY procedure, which runs in O.lg n/ time, is the key to maintaining the max-heap property.  The BUILD-MAX-HEAP procedure, which runs in linear time, produces a maxheap from an unordered input array.  The HEAPSORT procedure, which runs in O.n lg n/ time, sorts an array in place.  The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in O.lg n/ time, allow the heap data structure to implement a priority queue.  **Min Heap**  MIN-HEAPIFY -> Lowest value will be in top  BUILD-MIN-HEAP -> produces a min heap  HEAPSORT - Root has the lowest element, swap to the last position  The MIN-HEAP-INSERT, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY,  and HEAP-MINIMUM procedures, which run in O.lg n/ time, allow the heap  data structure to implement a priority queue.  Index with 1 -> parent = i/2,left = 2i,right =2i+1  Index with 0 parent = index / 2; odd parent = (index / 2) - 1;  l = (2 \* i) + 1;r = (2 \* i) + 2;  void increaseKey(int \*a, int index, int key){  if (a[index] > key) {  std::cout << "New Key is lesser than the earlier one\n";  return; } else { int parent; if (index % 2 == 1)  parent = index / 2; else parent = (index / 2) - 1;  std::cout << "\n parent:" << parent << "\n";  a[index] = key;while (index > 0 && a[parent] < a[index]) { std::cout << "\n Node :" << parent << "\n";  std::cout << "Index :" << index;  int x = a[index]; a[index] = a[parent];a[parent] = x;  index = parent;  if (index % 2 == 1) parent = index / 2;  else parent = (index / 2) - 1;} } }  void insertkey(int \*a, int key) {  //heapsize = heapsize + 1;  std::cout << "\nheapsize:" << heapsize;  a[heapsize] = -999; increaseKey(a, heapsize, key);//Max Heap -99999  }  void build\_maxheap(int \*a, int n)  { int i; for (i = n / 2; i >= 0; i--) { max\_heapify(a, i, n - 1); }}  The Heapsort takes O(nlogn) since the call to BUILD-MAX-HEAPIFY takes O(n) and the n-1 calls to MAX- HEAPIFY takes O(log n)  void extract\_max(int \*a){  int x = a[0];a[0] = a[heapsize - 1];a[heapsize - 1] = x;heapsize = heapsize - 1; max\_heapify(a, 0, heapsize - 1);}  INSERT.S; x/ inserts the element x into the set S, which is equivalent to the operation  MAXIMUM.S/ returns the element of S with the largest key.  EXTRACT-MAX.S/ removes and returns the element of S with the largest key. **O(logn)**  INCREASE-KEY.S; x; k/ increases the value of element x’s key to the new value k,which is assumed to be at least as large as x’s current key value. **O(logn)**  **Quick Sort : In place, not stable**  **Worst case:** O(n2). Let us assume the pivot element is always the right-most element: Input an already sorted list with n elements. So each partitioning leads to one list with n-1 elements and one list with 0 elements. Even if you choose the pivot element randomly, you can still be unlucky and always choose the maximum value in the list.  **Best case:** O(nlogn). If the pivot element is chosen in such way, that it partitions the list evenly:  int partition(int \*arr, int start, int end){  int pivot = arr[end];  int partitionIndex = start;  for (int i = start; i < end; i++){  if (arr[i] <= pivot){  int temp = arr[i];  arr[i] = arr[partitionIndex];  arr[partitionIndex] = temp;}}  int temp1 = arr[partitionIndex];  arr[partitionIndex] = arr[end];  arr[end] = temp1;return partitionIndex;}  void quicksort(int \*arr,int start,int end) {  if (start < end){  int pIndex = partition(arr, start, end);  quicksort(arr, start, pIndex - 1);  quicksort(arr, pIndex + 1, end);}} | **Shell Sort : Not Stable**  **Worst case performance O(n2)**  **Best case performance O(n log2 n)**  **Shell Sort:**The Shellsort is good for **Medium size array**. **Faster than insertion and selection**.problem Suppose a small item is on the far right, where the large items should be. To move this small item to its proper place on the left, all the intervening items (between the place where it is and where it should be) must be shifted one space right. So many swaps take place.The Shellsort achieves these large shifts by insertion-sorting widely spaced elements. After they are sorted, it sorts somewhat less widely spaced elements, and so on.  For example : there are 10 element, then d = d+1/2 = 5  If array index starts from 0, then compare 0th with 5th , else compare 1th with 6th  void ShellSort(std::vector <int >& num) {  for (size\_t i = 0; i < num.size(); ++i)  std::cout << num[i] << " ";  int i, temp, flag = 1, numLength = num.size();  int d = numLength; std::cout << "\n size" << d;  while (flag || (d > 1)) // boolean flag (true when not equal to 0) {  for (size\_t i = 0; i < num.size(); ++i)  std::cout << num[i] << " ";  flag = 0; // reset flag to 0 to check for future swaps  d = (d + 1) / 2;  for (i = 0; i < (numLength - d); i++)  {  std::cout << "\n " << num[i] << " " << num[i + d];  if (num[i + d] < num[i]) {  temp = num[i + d]; // swap positions i+d and i  num[i + d] = num[i];  num[i] = temp;  flag = 1; // tells swap has occurred } } } return; }   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **ALG** | **Bst** | **Avg** | **worst** | **stable** | **In-place** | | Bubble | n | n^2 | n^2 | yes | yes | | Insertion | n | n^2 | n^2 | yes | yes | | selection | n^2 | n^2 | n^2 | No | yes | | Shell | n | O((nlog(n))^2) | O((nlog(n))^2) | No | yes | | merge | nlogn | nlogn | nlogn | yes | No | | Quick | nlogn | nlogn | n^2 | No | yes | | counting | n+k | n+k | N^2 | yes | No | | Radix | nk | Nk | nk | yes | No | | Bucket | n+k | n+k | n^2 | yes | No | | BST-Insert |  | O(lgn) | O(n) |  |  | | BST-Search |  | O(lgn) | O(n) |  |  | | BST-Delete |  | O(lgn) | O(n) |  |  | | BST-Space |  | O(n) | O(n) |  |  | | AVL-Insert |  |  | O(lgn) |  |  | | AVL-Delete |  |  | O(lgn) |  |  | | AVL-Search |  |  | O(lgn) |  |  | | Splay-Insert |  | O(lgn) | Amortized (log n) |  |  | | SplayDelete |  | O(lgn) | amortized O(log n) |  |  | | SplaySearch |  | O(lgn) | amortized O(log n) |  |  | | RB/Btree-Ins |  | O(lgn) | O(lgn) |  |  | | RB/Btree-del |  | O(lgn) | O(lgn) |  |  | | RB/Btree-searc |  | O(lgn) | O(lgn) |  |  |   #include <iostream>  class queue{  int front, rear, maxsize; // \*que;  public: int \*que; queue( int size) {  front = 0; rear = 0; maxsize = size;  que = new int[10]; }    void enqueue( int val)  { if ((rear == (front - 1)) || (rear == maxsize && front == 0))  {std::cout << "\nqueue is full";return ;}  if (rear == maxsize) rear = 0;  que[rear] = val;  rear = rear+1; }  void dequeue() { if (front == rear)  { std::cout << "\nqueue is under flow";  return;  }  if (front+1 == maxsize)  front = 0;  else front = front + 1;  }  void display() {  //std::cout << "bb:" << que[0];  for ( int i = front; i != rear; i++){  if (i == maxsize) i = 0;  std::cout << que[i];  } } };  Radix Sort:  In computer science, radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.  170, 045, 075, 090, 002, 024, 802, 066  The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:  2 (bucket size for digits of 0: 170, 090)  2 (bucket size for digits of 2: 002, 802)  1 (bucket size for digits of 4: 024)  2 (bucket size for digits of 5: 045, 075)  1 (bucket size for digits of 6: 066)  A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:  2 (bucket size for digits of 0: 002, 802)  1 (bucket size for digits of 2: 024)  1 (bucket size for digits of 4: 045)  1 (bucket size for digits of 6: 066)  2 (bucket size for digits of 7: 170, 075)  1 (bucket size for digits of 9: 090)  A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:  6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)  1 (bucket size for digits of 1: 170)  1 (bucket size for digits of 8: 802) |
| **Stable : Bucket Sort, Bubble Sort, Merge Sort, Radix Sort, Counting Sort, Radix Sort, Insertion Sort**  **Tree traversals:**  **Pre-order: NLR In-order: LNR Post-order: LRN**  **Trees:**  \*Free Tree is a connected acyclic, undirected graph.  \***Tree** **Properties**: -Any two vertices are connected by a unique single path -G is acyclic and connected. -Forest is a collection of trees  \* |E| = |V| -1  **\**Rooted Trees:*** A tree is a finite set T of one or more nodes where one element called the root of T is distinguished. The remaining nodes in the tree form m disjoint subset (T1,.. Tm). (m>=0) where each subset is a tree  \****Ancestors*, *Descendents*:** Consider a node x in a rooted tree T with root r. We call any node y on the unique simple path from r to x an ancestor of x. If y is an ancestor of x, then x is a descendant of y. Every node is both an ancestor and a descendant of itself.) If y is an ancestor of x and x ¤ y, then y is a proper ancestor of x and x is a proper descendant of y.  \****Parent***, ***Child***: If the last edge on the simple path from the root r of a tree T to a node x is (y,x) then y is the parent of x, and x is a child of y.  \****Siblings***: If two nodes have the same parent, they are siblings.  \****External*** ***Node***: A node with no children is a leaf or external node.  \****Internal*** ***Node***: A non leaf node is an internal node.  \****Ordered* *Tree***: In ordered tree is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, and a kth child.  \***Degree** of a Node x: The number of children of a node x in a rooted tree T equals the degree of x.  \***Depth** of a Node x: The length of the simple path from the root r to a node x is the depth of x in T .  \***Level** of a Tree: A level of a tree consists of all nodes at the same depth.(Root is of level zero).  \***Height** of a Node : The height of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root  \***Binary Trees**: Tree Properties :Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y:*key* < x: *key*. If y is a node in the right subtree of x, then y:*key* > x:*key*.  We define binary trees recursively. A binary tree T is a structure defined on a finite set of nodes that either 1) contains no nodes, or 2) is composed of three disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.  **Operations Performed**:  (Search, minimum, maximum, predecessor, successor, insert and delete)  **Types**  **\**Full Binary Tree***: Each node is either a leaf or has degree exactly 2.\****Complete Binary Tree*** *A* complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.\****Perfect Binary Tree*** *A* perfect binary tree is a full binary tree in which all leaves are at the same depth or same level, and in which every parent has two children.\****Balanced Binary Tree*** A balanced binary tree is commonly defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1  **Formula**:  *Maximum* *nodes* at *level* i is 2^i *Max* no. of *nodes* in Binary-tree with *height* h is 2^(h+1)-1 *height* of *perfect* *Binary-tree* with n nodes h=lg(n+1)-1 *height* of *complete* *Binary*-tree= floor lgn for Binary-tree with n nodes, *min* *height*= floor lgn *max* *height*=n-1  **Pre-Order:**  void **preorder**(Node \*root) { if(root==NULL) return; cout<< root->data; preorder(root->left); preorder(root->right); }  **Thread in-order travasal:** struct Node\* **leftmost**(Node\* n) { if(n==NULL) return NULL; while(n->left!=NULL)n=n->left; return n; } void **inorder**(struct Node\* root) { sruct Node \*cur=leftmost(root); while(cur!=NULL) { cout<<cur->data; if(cur->rightThread) cur=cur->right; else cur=leftmost(cur->right); } }  **Insertion**:  struct node\* insert(struct node\* node, int key){ if (node == NULL) return newNode(key); if (key < node->key) node->left = insert(node->left, key); else node->right = insert(node->right, key); } **Min Value:** struct node \* minValueNode(struct node\* node){ struct node\* current = node; while (current->left != NULL) current = current->left;  return current;}  **Successor:**  We break the code for TREE-SUCCESSOR into two cases.  1. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x’s, right subtree, which we find in line 2 by calling TREE-MINIMUM .x:*right*/ .  2. On the other hand, If the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose  left child is also an ancestor of x.  **TREE-SUCCESSOR**(x)   if x:*right* != NIL   return TREE-MINIMUM (x:*right*)   y = x:*p*   while y != NIL and x == y:*right*   x = y   y = y:*p*  return y  **Deletion**  1. If z has no left child and right child, we just delete the node z 2. If z has just one child, which is its left child or right child, then we replace z by child. 3. otherwise If z has both left and right child, we find z's successor y which lies in z's right subtree and has no left child, we want to splice y out of its current location and have it replace z in the tree a.  If y is z’s right child , then we replace z by y, leaving y’s right child alone. b. Otherwise, y lies within z’s right subtree but is not z’s right child. In this case, we first replace y by its own right child, and then we replace ´ by y. | **Height:**  int height(struct node\* node) {    if (node==NULL) return 0;    else  {      int lHeight = height(node->left);      int rHeight = height(node->right);         return (lHeight > rHeight)? (lHeight+1): (rHeight+1);    } }  **Number of Nodes**  int BinaryTree::size(Node \*leaf, int& count=0) const { if(leaf != NULL)//if we are not at a leaf { count++; size(leaf->getLeft(), count);//recurisvly call the function and increment the count size(leaf->getRight(), count); }  return count;//return the count }  **Balanced Binary Trees:[AVL Trees]**  In BST the worst case running time for 3 operations is O(h) where h is the height of tree. h can be as small as lg(n) or as large as n-1, so BBT is required. And also avg path length of B.ST is approx. 2lg(n)  An **AVL** tree1 is a self-balancing binary search tree (BST) in which the heights of the two child subtrees of any node differ by at most one. Rotation : BF h(left) - h(right). if BF is not equal to 1 or -1 rotation is required. LL: 1)b becomes root, a takes b’s left child as right, b takes a as its left child. RR: 2)B becomes new root, c takes ownership of b’s right child as its left child B takes ownership of c as its right child. LR: Right subtree is left heavy, perform right rotation on right sub tree and then left rotation RL: Left subtree is right heavy, perform left rotation on left sub tree and right next.  **Splay Trees:**  A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.  **Splay** **Insertion** :  1. Insert the newly Inserted element similar to binary search tree.  2. Splay the newly inserted node to root  **Splay** **Tree** **Deletion**  One method of Splay Tree Deletion is as follows:  1. Splay the node to be deleted to the top  2. Make the left child the root  3. Make the left child's right subtree and attach it to the leftmost node of the right child  4. Make the right child the new right child of the new root  **Lookup:**  Accessing of a splay element—make the corresponding element as root, using Zig-Zag. Happy Zig-Zag!  **Multiway Trees:**  A multiway tree is a tree that can have more than two children. A multiway tree of order m (or an m-way tree) is one in which a tree can have m children, the nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and pointers to children.  **Properties:** 1. Each node has m children and m-1 keys  2. The keys in each node are in ascending order 3. The keys in the first i-1 children are smaller than the ith key  4. The keys in the last m-i-1 children are larger than the ith key  **B-Tree:**  1) The root has at least two sub trees unless it is a leaf. 2) Each non root and each non leaf node holds k-1 keys and k pointers to subtree where celing(m/2)<= k<=m 3) Each leaf node holds k-1 keys where celing(m/2)<=k<=m 4) All leaves are on same level  Operations in B-tree:  (Search, Insertion, deletion)  **Insertion**:  -A key is placed in a leaf that still has room - Insert Directly -A key is placed in a leaf that is full -> create a new node and move the middle node to parent  -A key is to be placed in a tree that is full -> split the nodes and create new nodes move the middle node to parent, If root doesnt have space split root and create internal nodes  **Deletion**: 1) Simple delete-> when after deleting the key from leaf, the node obeys cealing[m/2]-1 keys. 2) When the number of keys in the leaf are less than half, take its parent node and keep in the delete position and get the sibling 1st key into parent position if the sibling is having more than celing m/2-1 keys. 3) if the sibling has less than celing m/2 take 1 from parent key and the next sibling key and merge everything into single node and delete the right sibling. Case After combining if the parent is less than half full , create a new root from the parent, it’s sibling and old root. The lowest levels in the diagram are kept the same. 4) If we are deleting from non-leaf node, replace it with its immediate predissisor or successor.  **B \* Trees:**  B\* Trees are a variant of B-Tree. In B\* Trees, all nodes are required to be 2/3rd full, rather than half  -> The frequency of node splitting is decreased by delaying a split, when a node is full, **the keys in the full node are evenly distributed with its sibling**s. -> A split occurs when two siblings are full and the nodes are split into three -> The average utilization of B\* trees was found to 84% while B Tree is 69%.  **B+ Trees** In B-trees and B\* trees it is simple to write code to do an in-order traversal. However, if the blocks represent only one element from an internal node is accessed before switching to another disk block. This can be costly. In B+ trees, the leaves contain all the keys in tree(not just leave keys) and leaves have links forming a Linked list.  **Single Linked List**  **void insertAtLast(int p)**  { //Deletion at Last, Need to take care of the pointer  Node \*temp = new Node();  temp->data = p;  if (head == NULL) {  head = temp;  head->node = NULL;  return; }  Node \*first = head;  Node \*prev = first;  while (first != NULL) {  prev = first;  first = first->node; }  prev->node = temp; }  **void deleteAtLast() {**  if (head == NULL)  { return; }  Node \*first = head;  Node \*prev = first;  if (head->node == NULL) {  head = NULL;  return; }  while (first->node != NULL) {  prev = first; first = first->node;  } prev->node = NULL;  } | **Dequee**  **void insertAtFront(int val )** {          Node \*newNode = new Node();         newNode->data = val;          if (front == NULL) {                rear = newNode;         }  else {   newNode->next = front;         }   front = newNode; }  **void insertAtRear(int val )** {          Node \*newNode = new Node();         newNode->data = val;          if (rear == NULL) {                front = newNode;         }   else {                rear->next = newNode;         }  rear = newNode; }  **void deleteAtFront(){**          Node \*temp = front;          if (temp == NULL){                 return;         }   else {                temp = temp->next;                front = temp; } }  **void deleteAtLast(){**          Node \*temp = front;          Node \*temp1 = NULL;          if (temp == NULL){                 return;         }         else {                 while (temp->next != NULL) {                        temp1 = temp;                        temp = temp->next;                }                std::cout << "Temp ->data :" << temp->data<<"\n" ;                 if (temp1!= NULL)                        temp1->next = NULL;                 else front=temp1;                rear = temp1;       } }  **Swap:**  void swap(ListNode \*prev\_lhs, ListNode \*lhs, ListNode \*prev\_rhs, ListNode \*rhs) {  if (lhs == NULL || rhs == NULL ||  (prev\_lhs == NULL && prev\_rhs == NULL))  return; ListNode \*temp = rhs->next;  rhs->next = lhs->next; if (prev\_lhs != NULL)  prev\_lhs->next = rhs; lhs->next = temp; if (prev\_rhs != NULL)  prev\_rhs->next = lhs; }  void genOrder2Pattern() {  Node \* temp = head;  Node \*prev = new Node();  bool flag = true;  while (temp != NULL && temp->node != NULL) {  Node \*temp1 = temp->node;  temp->node = temp1->node;  prev->node = temp1;  prev = temp;  temp1->node = temp;  if (flag) { head = temp1;  flag = false; }  temp = temp->node; } }  void reverse() {  Node\* prev = NULL;  Node\* current = head;  while (current != NULL) {  Node\* next = current->node;  current->node = prev;  prev = current;  current = next;  }  head = prev;  }  **Doubly Linked List**  **void insertAtPos(int value,int pos) {** node \*cur = head; node \*prev = NULL; node \*newnode = new node(); newnode->data = value; newnode->rlink = NULL; newnode->llink = NULL;  int counter =0; while (cur != NULL) { if (counter == pos) { newnode->llink = prev; newnode->rlink = cur; prev->rlink = newnode; cur->llink = newnode; break; } else { prev = cur; cur = cur->rlink; counter++; } }}  **void deleteNum(int num){** node \* temp = head; node \*prev = NULL;  if (temp->rlink == NULL & temp->data == num){ head = NULL; return; } else if (temp->data == num){ temp = temp->rlink; temp->llink = NULL; head = temp; return; } while (temp->rlink != NULL){ if (temp->data == num){ prev->rlink = temp->rlink; temp->rlink->llink = prev; return; } else { prev = temp; temp = temp->rlink; } } std::cout << "dsdd:" << temp->data; if (temp->data == num) { tail = prev; tail->rlink = NULL; }  **Reverse :**  void reverse() {  tail = head;  Node \*curr = head; Node \* temp = NULL; Node \* next = NULL;  while (curr != NULL) {  temp = curr->llink; curr->llink = curr->rlink;  curr->rlink = temp; curr = curr->llink; } head = temp->llink; }  **void generateDLLSort2() {**  Node \*first = head;  Node \*prev = new Node();  bool flag = true;  while (first != NULL && first->rlink != NULL) {  Node \*temp1 = first->rlink;  Node \*temp2 = temp1->rlink;  first->rlink = temp1->rlink;  if (first->rlink != NULL) { //Check if the last node is having null , no need to point llink;  temp2->llink = first; }  prev->rlink = temp1;  temp1->llink = prev;  prev = first;  temp1->rlink = first;  first->llink = temp1;  Node \* cur = first->rlink;  if (flag){ head = temp1;  flag = false; } first = cur; }  tail = prev; head->llink = NULL; } | **-3-4 Trees** A 2-3-4 search tree is a tree that either is empty or has three types of nodes -2-nodes: with one key and two pointers -3-nodes:with two keys are 3 pointers -4-nodes: with three keys and 4 pointers.  **Red-black**  A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced. Each node of the tree now contains the attributes color, key, left, right, andp. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.  A red-black tree is abinary tree that satisﬁes the following red-black properties: 1. Every node is either red or black.  2. The root is black.  3. Every leaf (NIL) is black. 4. If a node is red, then both its children are black. 5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes. We use the **sentinel** so that we can treat a NIL child of a node x as an ordinary node whose parent is x. Although we instead could add a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well deﬁned, that approach would waste space. Instead, we use the one sentinel T:nil to represent all the NILs—all leaves and the root’s parent. Thevalues ofthe attributes p, left, right, and key of the sentinel are immaterial, although we may set them during the course of a procedure for our convenience.  A red-black tree with n internal nodes has height at most 2lg(n+1).  **Insertion:**  1)every incoming node is red. If the node colour and uncle colour(red) are same , then change the colour of G.P,P and uncle. 2) If the uncle node is black, do just rotation but in left subtree right rotation happens, change the col  our of P,G.P and uncle. Similarly if tn right subtree left rotation happens, change the colour. Or else no change.(LL, RR, LR, RR) 3) do this if only 2 connective red nodes occurs  **Digital Search Trees**  Digital Search Trees are Binary Search Trees that are guaranteed to have a relatively small height and require no balancing operations  -> The left child of a node has the next bit 0 -> The right child of a node has the next bit 1  The max height in a digital search tree is **O(log num\_key\_bits)** so the max search time is same  **Binary Tires:**  These allow inorder traversal whereas Binary doesn’t allow.  Information is in leaf nodes.  **R Trees:**  R-Trees are used to enable effecient search of 2D spatial data. The non-leaf nodes of a b-tree contain rectangle coordinates of child nodes. Insertion:done in a rectangle that needs the least enlargement. splitting on overflowing node: R\* tree topological split gives the best for spatial map application.Here, when a node is full, a portion of the nodes are removed and reinserted per level is allowed to prevent an infinite loop of overflows. |