|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Linked Lists** | | | | | | | | | | | | | | | | | | | | | | |
| Singly | | | | | | Doubly | | | Circular | | | | | | Reverse [O(*n*)] | | | | | Rec isSorted | | |
| struct node {  int data;  struct node\* next;  }; | | | | | | struct node{  int data;  struct node\* prev;  struct node\* next;  }; | | | * Pointer of the last node points to the first. * End of the list: when head->next == front. * Must keep a link between both. | | | | | | while (nextptr != NULL) {  cur->next = prev;  prev = cur;  cur = nextptr;  nextpr = nextpr->next;  } | | | | | int isSorted(struct node\* ptr) {  if (ptr == NULL) return 1;  while (ptr->next != NULL) {  if (ptr->data > ptr->next->data) return 0;  ptr = ptr->next;  }  return 1;  } | | |
| Creation and Init [O(1)] | | | | | | | | | | | | | | | Traversal [O(*n*)] | | | | | Access Data [O(1)] | | |
| struct node\* head = (struct node\*)malloc(sizeof(struct node));  //Empty list  head = NULL;  head->next = NULL;  head->prev = NULL;  struct node\* front = head;  struct node\* temp; | | | | | | | | | | | | | | | current = head;  while (current != NULL) //End of list  current = current->next; | | | | | current->data | | |
| Insert | | | | | | | | | | | | | | | | | | | | Delete [O(*n*)] | | |
| Insert to Front [O(1)] | | | | | | Insert to Back [O(*n*)] | | | | | | Insert In Order [O(*n*)] | | | | | | | | if(head != NULL) //Empty list; included before deletions  Before head->next is NULL.  if(head->next->data == x)  Point temp to head->next.  Point head->next to head->next->next.  Free temp.  Return front.  Iterate head to head->next.  Return front. | | |
| //Empty list  if (head == NULL) return current;  Point current->next to head.  Point head to current.  Return head. | | | | | | Iterate head before head->next is NULL.  Point head->next to current.  Return front. | | | | | | //Empty list or item at beginning  if (head == NULL || x < head->data)  return current;  Iterate head before head->next is NULL && before head->next->data > x.  Point current->next to head->next.  Point head->next to current.  Return front. | | | | | | | |
| Count Frequency | | | | | | RecSum All Vals | | | | | | RecInsert in Order | | | | | | | | RecDel | | |
| int freq(struct node\* mylist, int value){  int res = 0;  while (ptr != NULL) {  if (ptr->data == value) res++;  ptr = ptr->next;  }  } | | | | | | Base case: empty list  Return 0.  Return (ptr->data + sum(ptr->next)); | | | | | | Base case: node is at the front  if(head == NULL || x <= head->data)  Save x to temp->data.  Point temp->next to head.  Return temp.  Recursion  Point head->next to recInsert(head->next, x).  Return head. | | | | | | | | Simple case: empty list  if(head == NULL)  return NULL;  Base case  if(head->data == x){  Point temp to head->next.  Free head.  Return temp.  }  Recursion  Point head->next = recDel(head->next, x).  Return head. | | |
| **Queues** | | | | | | | | | | | | | | | | | | | | | | |
| Array | | | | | | | | | | | | | | Linked List | | | | | | | | |
| struct queue {  int\* elements;  int front;  int numElements;  int queueSize;  }; | struct queue\* myQ = (struct queue\*)malloc(sizeof(struct queue));  myQ->elements = (int\*)malloc(sizeof(int)\*INIT\_SIZE);  myQ->front = 0;  myQ->numElements = 0;  myQ->queueSize = INIT\_SIZE; | | | | | | | | | | | | | struct node {  int data;  struct node\* next;  };  struct queue {  struct node\* front;  struct node\* back;  }; | | | | struct queue\* myQ = (struct queue\*)malloc(sizeof(struct queue));  myQ->front = NULL;  myQ->back = NULL;  //Check that memory was allocated correctly struct node\* temp = (struct node\*)malloc(sizeof(struct node)); | | | | |
| Full, Empty, and Front | | | | | | | | | | | | | | | | | | | | | | |
| //Full  myQ->numElements == myQ->queueSize | | | | | //Empty  myQ->numElements == 0 | | | | | | | | | //Empty  myQ->front == NULL | | | | | | | //Front  myQ->front->data | |
| Enqueue | | | | | | | | | | | | | | | | | | | | | | |
| //O[(1)]  If not full  myQ->elements[(myQ->front + myQ->numElements) % myQ->queueSize] = x;  Increment myQ->numElements.  If full  //Reallocate more memory  myQ->elements = (int\*)realloc(myQ->elements, (myQ->queueSize)\*sizeof(int) \* 2);  //Copy values  for(i = [0, myQ->front)  myQ->elements[i+myQ->queueSize] = myQ->elements[i];  myQ->elements[i+myQ->queuesize] = x;  (myQ->queueSize) \*= 2;  Increment myQ->elements. | | | | | | | | | | | | | | //O[(*n*)]  Store x in temp->data.  Set temp->next to NULL.  If myQ->back != NULL, set myQ->back->next to temp.  Set myQ->back to temp.  If myQ->front == NULL, set myQ->front to temp. | | | | | | | | |
| Dequeue | | | | | | | | | | | | | | | | | | | | | | |
| //O[(*n*)]  If empty, return EMPTY.  Store myQ->elements[] at index front in retval.  Adjust myQ->front to (myQ->front + 1)%myQ->queueSize.  Decrement numElements.  Return retval. | | | | | | | | | | | | | | //O[(*1*)]  If empty, return EMPTY.  Store myQ->front->data in retval.  Set temp to myQ->front.  Set myQ->front to myQ->front->next.  If myQ->front == NULL, set myQ->back to NULL.  Free temp.  Return retval. | | | | | | | | |
| Breadth First Search [O(*n*)] | | | | | | | | | | | | | |  | | | | | | | | |
| * + - * A counter of steps starts at 0.       * Nodes visited in order.       * Each node enqueues adjacent spots that are valid and have not yet been visited.       * The node is dequeued and marked as visited by the number of steps to get to it. | | | | | | | | | | | | | |
| **Stacks** | | | | | | | | | | | | | | | | | | | | | | |
| Array | | | | | | | | | | | | | | Linked List | | | | | | | | |
| struct stack {  int items[SIZE];  int top;  } | struct stack\* stackptr = (struct stack\*)malloc(sizeof(struct stack));  stackptr->top = -1; | | | | | | | | | | | | | struct stack {  int data;  struct stack\* next;  }; | | | | | struct stack\* stackptr = (struct stack\*)malloc(sizeof(struct stack));  //Pass &stackptr into functions as a double-pointer  //Ensure memory was allocated dynamically before continuing  struct stack\* temp = (struct stack\*)malloc(sizeof(struct stack));  //Init  &(\*stackptr) = NULL; | | | |
| Full/Empty/Top | | | | | | | | | | | | | | | | | | | | | | |
| //Full  stack->top == SIZE - 1 | | //Empty  stackptr->top == -1 | | | | | //Top  If empty, return EMPTY.  Return items[] at top index. | | | | | | | //Empty  stackptr == NULL. | | | | | | | //Top  If empty, return -1.  Else, return stackptr->data. | |
| Push | | | | | | | | | | | | | | | | | | | | | | |
| //O[(*n*)]  If full, return 0;  Insert x into stackptr->items[] at index top.  Increment top index.  Return 1. | | | | | | | | | | | | | | //O[(*n*)]  Save x into temp->data.  Point temp->next to \*(&stackptr). | | | | | | | | |
| Pop | | | | | | | | | | | | | | | | | | | | | | |
| //O[(*1*)]  If empty, return -1.  Save items[] at top index to retval.  Decrement top index.  Return retval. | | | | | | | | | | | | | | //O[(*1*)]  Point temp to NULL.  Point temp to \*(&stackptr).  Point \*(&stackptr) to \*(&stackptr)->next.  Point temp->next to NULL.  Return temp. | | | | | | | | |
| Operand Stack | | | | | | | | | | | | | Infix to Postfix | | | | | | | | | |
| * Read an operand 🡪 push onto stack * Read an operator 🡪 pop last 2 items off the stack op2, followed by op1. Calculate op1 op op2 and push this value onto the stack.   + If you ever try to pop an empty stack, the expression is invalid postfix expression.   + If after running operations you end up with a stack size > 1, it’s also invalid.     - Should only end up with 1 answer. | | | | | | | | | | | | | * Open parenthesis 🡪 push onto stack * Operand 🡪 place into expression in order. * Close parenthesis 🡪 pop items off stack, placing each in the expression until we hit the first open parenthesis. * Operator 🡪 pop off the stack each operator of equal or higher precedence, placing each into the expression.   + Stop popping when you reach…     - an operator of lower precedence.     - a parenthesis.     - the end of the stack.   + Push this operator onto the stack. * End 🡪 pop off remaining operators and place in the expression. | | | | | | | | | |
| **Binary Trees** | | | | | | | | | | | | | | | | | | | | | | |
| Implementation | | | | | | | | | | Traversal (O(*n*)) | | | | | | | | | | | | |
| struct tree\_node {  int data;  struct tree\_node\* left;  struct tree\_node\* right;  }; | | | struct tree\_node\* root = (struct tree\_node\*)malloc(sizeof(struct tree\_node));  root->data = VALUE;  root->left = NULL;  root->right = NULL;  struct tree\_node\* element;  struct tree\_node current\_ptr; | | | | | | | Inorder (Straight down) | | | | | | | Preorder (Root 🡪each until far L, R) | | | | | Postorder (LR leaf nodes, Root) |
| if (current\_ptr != NULL) {  inorder(current\_ptr->left);  //access current\_ptr->data  inorder(current\_ptr->right);  } | | | | | | | if (current\_ptr != NULL) {  //access current\_ptr->data  preorder(current\_ptr->left);  preorder(current\_ptr->right);  } | | | | | if (current\_ptr != NULL) {  preorder(current\_ptr->left);  preorder(current\_ptr->right);  //access current\_ptr->data  } |
| Search [O(*n*)] | | | | | | | | | | Insertion [O(*n* 🡪log2*n*)] | | | | | | | | | | | | |
| If empty, return element.  //Go left  If (x < root->data)  Return search(root->left, x);  //Go right  If (x>root->data)  Return search(root->right, x).  Return 1. | | | | | | | | | | if empty, return element.  //Item should be inserted to the right  if (element->data > root->data)  if (root->right != NULL)  Set root->right to insert(root->right, element);  else  Set root->right to element.  //Item should be inserted to the left  else  If(root->right != NULL)  Set root->left to insert(root->left, element).  else  Set root->left to element.  Return root. | | | | | | | | | | | | |
| **AVL Trees**: height of left child differs from height of right child by at most 1. {Recurrence: SH = SH-1 + SH-2 + 1} | | | | | | | | | | | | | | | | | | | | | | |
| Insertion [O(*n* log *n*)] | | | | | | | | | | Delete [O(*n* log *n*)] | | | | | | | | | | | | |
| 1. Do a normal binary tree insert. 2. Restore the tree based on this leaf node.    1. Calculate the heights of the left and right subtrees, use this to set the potentially new height of the node.    2. If they are within one of each other, recursively restore the parent node.    3. If not, then perform the appropriate restructuring described above on that particular node, THEN recursively call the method on the appropriate parent node. | | | | | | | | | | 1. Do a normal binary tree delete. 2. Restore the tree based on this leaf node, where multiple nodes may need to be rebalanced.   a) If one side is longer than the other, choose that side.  b) If the two sides are equal, go to the same side as the parent is to the grandparent. | | | | | | | | | | | | |
| **Binary Search Applications** | | | | | | | | | | | | | | | | | | | | | | |
| //Crystal Etching  double findTime(double f1, double f2, double a, double b, double c) {  double targettime = (f2-f1)/(f1\*f2);  double low = 0;  double high = targettime/a;  double mid = 0;  while (high - low > EPSILON) {    mid = (high+low)/2;  double ans = f(a,b,c,mid);    if (ans<targettime)  low = mid;  else if (ans == targettime)  break;  else  high = mid;  }  return mid;  }  double f(double a, double b, double c, double t) {  return a\*t+b\*(1-exp(-c\*t));  } | | | | | | | | | | | //A Careful Approach: Landing Planes  double getMaxTime(int\* perm, struct interval\* times, int length) {  // Set up our binary search.  double high = getMaxInterval(times, length);  double low = 0, mid = (low+high)/2;  // Keep going until our interval is tiny.  while (high-low > 1e-9) {    mid = (low+high)/2;    // Standard binary search.  if (works(perm,times,length,mid))  low = mid;  else  high = mid;  }  return mid;  } | | | | | | | | | | | |
| **Hash Table Probing** | | | | | | | | | | | | | | | |  | | | | | | |
| Overwrite data | | | | Linear probing | | | | Quadratic probing | | | | | | | |
| Lookup, Insert, Delete [O(*n*)] | | | | f(x+1)%TABLE\_SIZE | | | | f(x + i^2)%TABLE\_SIZE | | | | | | | |
| Linear chaining hashing: array of LL. | | | | Insert [O(1) 🡪 O(*n*)] | | | | int slot = f(x);  int tryval = slot, i = 1;  while(array[tryval]!=NULL) {  tryval = (slot + i\*i)%SIZE;  i++;  } | | | | | | | |
|  | | | | | | | |