TAVE 서기

|  |  |  |  |
| --- | --- | --- | --- |
| **서기 내용** | | | |
| **서기 일자** | 20.08.23 | **서기** | 최인아 |
| **주제** | 자료구조 복습 | | |
| **시간** | 15:00~17:00 | **장소** | MOIM 1호점 |
| **스터디**  **인원** | 방수영 송나은 최인아 양효빈 차영훈 | | |
| **내용** | | | |
| **배운 내용** | 1. 연결리스트  2. 스택  3. 큐  4. 힙  5. 트리 | | |
| 1. 연결리스트  1-1. 배경  - 배열은 삭제하고 뒤에 모든 값을 옮겨야 되서 O(n)만큼의 시간이 걸린다.    1-2. 단일 연결 리스트  - 링크를 가진 노드들의 순열  - 구성  -> Data : 실제 자료를 저장  -> Link : 다음 노드에 대한 주소 저장 (포인터)  - 삽입/삭제가 간단  - 최대 자료의 개수가 미리 정해지지 않음    1-2-1. 단일 연결 리스트 - search  - ptr 부터 시작해서 link를 타고 찾고자하는 data가 나올때까지 탐색  - 시간 복잡도 O(n)    1-2-2. 단일 연결 리스트 - insert  - 원래있던 연결을 끊고 새로운 노드를 삽입하고 link를 다시 정해줌  - 시간 복잡도 O(1)    1-2-3. 단일 연결 리스트 - delete  - 원래있던 연결을 끊고 삭제한 노드 앞뒤로 link를 다시 정해줌  - 시간 복잡도 O(1)    1-3. 더미 헤드  - Insert를 할때 중간에 추가하냐, 제일 앞에 추가하냐에 따라서 코드가 달라짐    - 제일 앞에 헤드를 더미로 두고 시작을 더미 헤드 다음부터로 하면  - 코드를 따로 짜지 않아도 한번에 해결 가능!    1-4. 응용  - 이런식으로 데이터를 여러 개 해도 되고    - 이런식으로 Link를 여러 개 해서 순서를 여러 개 할수도 있고    - 암튼 다양하게 활용하면 꽤나 유용하다!  1-5. 다중 연결 리스트  - 링크를 양방향으로 둬서 양방향으로 이동 가능하게 한 것    1-6. 원형 다중 연결 리스트  - 다중 연결 리스트에서 제일 끝과 앞을 연결 한 것    1-7. C++ 코드 구현  - 더미헤드가 있는 단일 연결 리스트  - 간단한 스택을 구현한다.    #include <iostream>  using namespace std;  template <typename T>  class Node {  public:      T data;      Node<T>\* link;      Node() : link(nullptr) { }      Node(T \_data, Node<T>\* \_link) : data(\_data), link(\_link) { }      ~Node() { delete link; }  };  template <typename T>  class List {  private:      Node<T>\* head;  public:      List() { head = new Node<T>(); }      void push(T data) {          Node<T>\* ptr = new Node<T>(data, head->link);          head->link = ptr;          size++;      }      void pop() {          if (!empty()) {              Node<T>\* ptr = head->link;              head->link = head->link->link;              ptr->link = nullptr;              delete ptr;          }      }      bool empty() { return head->link == nullptr; }      ~List() { delete head; }      template <typename K>      friend ostream& operator<<(ostream& os, List<K>& self);  };  template <typename T>  ostream& operator<<(ostream& os, List<T>& self) {      Node<T>\* p = self.head;      cout << "[";      while(p->link != nullptr) {          p = p->link;          cout << p->data << " ";      }      cout << "]";      return os;  }  int main() {      List<int> l;      l.push(1); cout << l << endl;      l.push(2); cout << l << endl;      l.push(3); cout << l << endl;      l.push(4); cout << l << endl;      l.push(5); cout << l << endl;      l.pop(); cout << l << endl;      l.pop(); cout << l << endl;      l.pop(); cout << l << endl;      l.push(9); cout << l << endl;      l.pop(); cout << l << endl;      l.push(10); cout << l << endl;      return 0;  }    1-8. 실제 쓰임새  - List 헤더에 다중 연결 리스트가 구현되어있다.  #include <iostream>  #include <list>  using namespace std;  int main(){      list<int> l;      l.push\_back(5); // 뒤에 추가      l.pop\_back(); // 뒤에 제거      l.push\_front(4); // 앞에 추가      l.pop\_front(); // 앞에 제거      cout << l.front(); // 앞에 노드 값      cout << l.back(); // 뒤에 노드 값      cout << l.size(); // 리스트 크기      cout << l.empty(); // 리스트가 비었는지 확인  }  1-9. 관련 문제  - https://www.acmicpc.net/problem/1158  예제 입력 7 3  예제 출력 <3, 6, 2, 7, 5, 1, 4>    #include <iostream>  #include <list>    using namespace std;    int main() {      int N, K;      list<int> l;        cin >> N >> K;        for (int i = 1; i <= N; i++)          l.push\_back(i);    cout << "<";      while (l.size()) {          if (l.size() == 1){              cout << l.front() << ">";              l.pop\_front();              break;          }          for (int i = 1; i < K; i++) {              l.push\_back(l.front());              l.pop\_front();          }          cout << l.front() << ", ";          l.pop\_front();      }      cout << endl;      return 0;  }  2. 스택  2-1. 개념  - 입구와 출구가 하나인 자료구조  -> 밑이 막힌 긴 통  -> LIFO (Last In First Out)  2-2. 연산  - Push : 스택에 데이터를 추가함  - Pop : 스택에서 가장 최근에 추가된 데이터를 빼냄  - GetTop : 스택에서 빠질 값을 얻어냄  - Initialization/Destruction : Constructor/Destructor & RemoveAll  2-3. 코드 구현      #include <iostream>  #include <string.h>  using namespace std;  int stack[10001], top = -1;  // 1. 메서드 //  // ——————————————————  void push(int x){  stack[++top] = x;  }  int empty(){  if(top < 0)  return 1;  else  return 0;  }  void pop(){  if(empty() == 1)  cout << "-1" << "\n";  else {  cout << stack[top] << "\n";  stack[top——————————————————] = 0;  }  }  // 2. main  // ——————————————————  int main(void) {  int n;  cin >> n;  for(int i=0;i<n;i++){  char str[10];  cin >> str;    if (!strcmp(str,"push")){  int x;  cin >> x;  push(x);  }  else if (!strcmp(str, "top")){  if (empty() == 1)  cout << "-1" << "\n";  else  cout << stack[top] << "\n";  }  else if (!strcmp(str, "size")){  cout << top+1 << "\n";  }  else if (!strcmp(str, "empty")){  cout << empty() << "\n";  }  else if (!strcmp(str,"pop")){  pop();  }  }  return 0;  }  3. 큐  3-1. 정의  - 큐 : FIFO(First-in, First-out) 구조의 자료구조  예) 줄서기  3-2. 기본 연산  void QueueInit(Queue \* pq);  - 큐의 초기화를 진행한다.  - 큐 생성 후 제일 먼저 호출되어야 하는 함수이다.  int QIsEmpty(Queue \* pq);  - 큐가 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.  void Enqueue(Queue \* pq, Data data);  - 큐에 데이터를 저장한다. 매개변수 data로 전달된 값을 저장한다.  Data Dequeue(Queue \* pq);  - 저장순서가 가장 앞선 데이터를 삭제한다.  - 삭제된 데이터는 반환된다.  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.  Data QPeek(Queue \* pq);  - 저장순서가 가장 앞선 데이터를 반환하되 삭제하지 않는다.  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.  3-3-1. 큐의 배열 기반 구현 - enqueue  - Front : 큐의 머리를 의미 (삭제 발생)  - Rear : 큐의 꼬리를 의미 (삽입 발생)  - R을 한 칸 이동시킨 후 새로운 데이터를 저장    3-3-2. 큐의 배열 기반 구현 - dequeue  - F가 가리키는 데이터를 반환 후 F를 한 칸 이동시킨다.  - 문제점 : 앞쪽의 배열은 비어있지만 R을 오른쪽으로 한 칸 더 이동시킬 수가 없어서 더 이상 데이터를 추가할 수가 없다.  - 해결책 : 원형 큐    3-4. 원형 큐  - 정의 : 배열의 머리와 끝을 연결한 구조  - 연산  -> enqueue : 배열기반 큐와 마찬가지로 R이 이동한 다음에 데이터를 저장  -> dequeue : 배열기반 큐와 마찬가지로 F가 가리키는 데이터 반환 후 F이동    - 문제점  - 큐가 Full or empty인 경우 F와 R의 위치만으로는 구분할 수 없다.    - 문제 해결  -> 저장 공간 하나를 잃고 해결  -> 데이터가 하나 존재하는 경우 F와 R이 같은 위치를 가리켰는데, 초기화 직후에 밑쪽의 상태가 되게 한다.  -> 하나를 비운 상태에선 Full로 인정    3-5. 구현            3-6. 원형 큐의 구조  - 초기 공백 상태 : front = rear = 0  - front와 rear의 위치가 배열의 마지막 인덱스 n-1에서 논리적인 다음 자리인 인덱스 0번으로 이동하기 위해서 나머지연산자 mod를 사용  -> 3 + 4 = 0 ...3 (몫 = 0, 나머지 = 3)  -> 3 mod 4 = 3  - 순차 큐와 원형 큐의 비교   |  |  |  | | --- | --- | --- | | 종류 | 삽입 위치 | 삭제 위치 | | 순차 큐 | rear = rear + 1 | front = front + 1 | | 원형 큐 | rear = (rear + 1) mod n | front = (front + 1) mod n |   - 사용조건) 공백 상태와 포화 상태 구분을 쉽게 하기 위해서 front가 있는 자리는 사용하지 않고 항상 빈자리로 둠  - 초기 공백 원형 큐 생성 알고리즘  -> 크기가 n인 1차원 배열 생성  -> front와 rear를 0 으로 초기화  - 공백 원형 큐 생성  createQueue()  cQ[n];  front <- 0;  rear <- 0;  end createQueue()  - 원형 큐의 공백 상태 검사  isEmpty(cQ)  if (front = rear) then return true;  else return false;  end isEmpty()  - 원형 큐의 포화 상태 검사  isFull(cQ)  if(((rear + 1) mod n) = front) then return true;  else return false;  end isFull()  - 원형 큐의 상태에 따른 front와 rear의 관계   |  |  | | --- | --- | | 구분 | 조건 | | 공백 상태 | front = rear | | 포화 상태 | (rear + 1) mod n = front |   - 원형 큐의 삽입 알고리즘  -> rear의 값을 조정하여 삽입할 자리를 준비 :  rear <- (rear + 1) mod n;  -> 준비한 자리 cQ[rear]에 원소 item을 삽입  - 원형 큐의 원소 삽입  enQueue(cQ, item)  if (isFull(cQ)) then Queue\_Full();  else {  rear <- (rear + 1) mod n;  cQ(rear) <- item;  }  end enQueue()  - 원형 큐의 삭제 알고리즘  -> front의 값을 조정하여 삭제할 자리를 준비  -> 준비한 자리에 있는 원소 cQ[front]를 삭제하여 반환  deQueue(cQ)  if (isEmpty(cQ)) then Queue\_Empty();  else {  front <- (front - 1) mod n;  return cQ[front];  }  end deQueue()  4. 힙  4-1. 개념  - 완전 이진트리의 일종으로 우선순위 큐를 위하여 만들어진 자료구조   * 여러 개의 값들 중에서 최댓값이나 최솟값을 빠르게 찾아내도록 만들어진 자료구조 * 큰 값이 상위 레벨에 있고 작은 값이 하위 레벨에 있다는 정도 * 간단히 말하면 부모 노드의 키 값이 자식 노드의 키 값보다 항상 큰(작은) 이진트리 * 힙 트리에서는 중복된 값을 허용 (이진 탐색 트리에서는 중복된 값을 허용하지 않음) * n개의 노드를 가지고 있는 힙의 높이 : O(logn)   4-2. 종류   * 최대 힙 (max heap) : 부모 노드의 키 값이 자식 노드의 키 값보다 크거나 같은 완전 이진 트리   -> key(부모 노드) >= key(자식 노드)   * 최소 힙 (min heap) : 부모 노드의 키 값이 자식 노드의 키 값보다 작거나 같은 완전 이진 트리   -> key(부모 노드) <= key(자식 노드)  4-3. 구현   * 힙을 저장하는 표준적인 자료구조는 배열 * 구현을 쉽게 하기 위하여 배열이 첫 번째 인덱스인 0은 사용하지 않음 * 특정 위치의 노드 번호는 새로운 노드가 추가되어도 변하지 않음 * 부모 노드와 자식 노드의 관계   -> 왼쪽 자식의 인덱스 = (부모의 인덱스) \* 2  -> 오른쪽 자식의 인덱스 = (부모의 인덱스) \* 2 + 1  -> 부모의 인덱스 = (자식의 인덱스) / 2    4-4. 삽입  - 힙에 새로운 요소가 들어오면, 일단 새로운 노드를 힙의 마지막 노드에 이어 삽입  - 새로운 노드를 부모 노드들과 교환해서 힙의 성질을 만족      insert\_max\_heap(A, key)  heap\_size = heap\_size + 1;  i = heap\_size;  A[i] = key;  while i != 1 and A[i] > A[PARENT(i)] do  A[i] <-> A[PARENT];  i = PARENT(i);  4-5. 힙 삭제   1. 최대 힙에서 최댓값은 루트 노드이므로 루트 노드가 삭제됨, 최대 힙(max heap)에서 삭제 연산은 최댓값을 가진 요소를 삭제 2. 삭제된 루트 노드에는 힙의 마지막 노드를 가져옴 3. 힙 재구성       delete\_max\_heap(A)  item = A[1];  A[1] = A[heap\_size];  heap\_size = heap\_size – 1;  i = 2; //parent = 1; child = 2;  while i <= heap\_size do  if i < heap\_size and A[LEFT(i)] > A[RIGHT(i)]  then largest = LEFT(i);  else largest = RIGHT(i);  if A[PARENT(largest)] > A[largest]  then break;  A[PARENT(largest)] <-> A[largest];  i = CHILD(largest);  return item;  5. 트리  5-1. 개념  - 트리는 노드로 이루어진 자료 구조이다.  - 트리는 하나의 루트 노드를 갖는다.  - 루트 노드는 0개 이상의 자식 노드를 갖고 있다.  - 그 자식 노드 또한 0개 이상의 자식 노드를 갖고 있고, 이는 반복적으로 정의된다.  5-2. 용어    - 루트 노드(root node): 부모가 없는 노드, 트리는 하나의 루트 노드만을 가진다.  - 단말 노드(leaf node): 자식이 없는 노드, ‘말단 노드’ 또는 ‘잎 노드’라고도 부른다.  - 내부(internal) 노드: 단말 노드가 아닌 노드  - 간선(edge): 노드를 연결하는 선 (link, branch 라고도 부름)  - 형제(sibling): 같은 부모를 가지는 노드  - 노드의 크기(size): 자신을 포함한 모든 자손 노드의 개수  - 노드의 깊이(depth): 루트에서 어떤 노드에 도달하기 위해 거쳐야 하는 간선의 수  - 노드의 레벨(level): 트리의 특정 깊이를 가지는 노드의 집합  - 노드의 차수(degree): 하위 트리 개수 / 간선 수 (degree) = 각 노드가 지닌 가지의 수  - 트리의 차수(degree of tree): 트리의 최대 차수  - 트리의 높이(height): 루트 노드에서 가장 깊숙히 있는 노드의 깊이  - 포화 이진 트리(Full Binary Tree) : 모든 레벨이 꽉 찬 이진 트리  - 완전 이진 트리(Complete Binary Tree) : 포화 이진 트리처럼 모든 레벨이 꽉 찬 상태는 아니지만, 차곡차곡 빈틈 없이 노드가 채워진 이진 트리  5-3. 이진 탐색 트리 - 삽입 알고리즘  삽입이란 특정 위치에 원하는 정보를 넣는 행위를 뜻한다. 그렇다. 특정 위치를 먼저 알아내야지만이 그 위치에 데이터를 삽입할 수 있다. 또한 내가 삽입하려는 데이터가 이미 존재하는 것일 수도 있기 때문에, 먼저 탐색을 해야만 한다. 결과적으로 탐색을 실패한 위치가 바로 새로운 노드를 삽는 위치가 된다.  - 트리 노드 삽입 코드(c++).  void Insert(TreeNode\*& tree, ItemType item)  {  if (tree == NULL)  {  tree = new TreeNode;  tree->right = NULL;  tree->left = NULL;  tree->info = item;  }  else if (item < tree->info)  Insert(tree->left, item);  else  Insert(tree->right, item);  }  5-4. 이진 탐색 트리 - 삭제 알고리즘  삭제도 먼저 삽입과 마찬가지로 탐색을 먼저 해야한다. 삭제하길 원하는 키 값이 트리 안의 어디에 위치하는지를 알아야 삭제할 수 있기 때문이다. 노드를 탐색했다면, 아래의 3가지 경우를 고려해야 한다. 삭제시 발생할 수 있는 상황 3가지다.  - 삭제하려는 노드가 단말노드일 경우    단말 노드를 삭제하는 행위는 매우 간단하다. 부모 노드를 찾아서 부모 노드 안의 해당 링크(그림의 경우 오른쪽 링크) 필드를 NULL로 만들어서 연결을 끊어주면 된다. 동적으로 할당된 노드였을 경우, 메모리 반납을 하고 링크필드를 NULL로 만들어주면 된다.  - 삭제하려는 노드가 하나의 서브트리만 가지는 경우    68을 가지고 있는 노드를 삭제하고자 한다. 이것 또한 간단한데, 35를 가지고 있는 부모 노드의 오른쪽 링크 필드가 99를 가리키게 수정하면 된다. 동적할당된 노드였다면, 68을 먼저 반납해주고 부모의 오른쪽 링크에 99를 붙여주면 된다.  - 삭제하려는 노드가 두 개의 서브트리를 모두 가지고 있는 경우    18을 삭제하려 한다. 삭제한 이후에 35를 가지고 있는 부모 노드의 왼쪽 링크필드를 어떤 노드에 연결시켜야 할까? 결론부터 말하면 왼쪽 서브트리에 있는 값 중 가장 큰 값, 혹은 오른쪽 서브트리에 있는 값 중 가장 작은 값을 가지고 있는 노드를 연결시켜주면 된다. 22를 선택했다고 가정해보자. 그러면 우리가 할 일은 삭제되는 노드의 오른쪽 서브 트리에서 왼쪽 자식 링크를 타고 NULL이 나올 때까지 게속 진행하면 된다. 그렇게 찾아낸 값을 35의 왼쪽 링크 필드에 이어주고 22의 왼쪽 링크필드를 7에, 오른쪽 링크필드를 26에 연결해주면 된다.  - 트리 노드 삭제 알고리즘(c++)  void Delete(TreeNode\*& tree, ItemType item)  // Deletes item from tree.  // Post: item is not in tree.  {  if (item < tree->info)  Delete(tree->left, item); // Look in left subtree.  else if (item > tree->info)  Delete(tree->right, item); // Look in right subtree.  else  DeleteNode(tree); // Node found; call DeleteNode.  }  void DeleteNode(TreeNode\*& tree)  {  ItemType data;  TreeNode\* tempPtr;  tempPtr = tree;  if (tree->left == NULL)  {  tree = tree->right;  delete tempPtr;  }  else if (tree->right == NULL)  {  tree = tree->left;  delete tempPtr;  }  else  {  GetPredecessor(tree->left, data);  tree->info = data;  Delete(tree->left, data); // Delete predecessor node.  }  }  void GetPredecessor(TreeNode\* tree, ItemType& data)  // Sets data to the info member of the right-most node in tree.  {  while (tree->right != NULL)  tree = tree->right;  data = tree->info;  } | | |
| **과제할당** | 각자 맡은 부분 개념 발표  -> 버블 정렬/선택 정렬, 삽입 정렬, 병합 정렬, 힙 정렬, 퀵 정렬  각자 맡은 정렬에 대한 문제 풀기  -> 2750 - 삽입, 선택, 버블  -> 2751 - 병합, 퀵, 힙 | | |
| **특이사항** |  | | |
| **비고** |  | | |