DataStructure

20195023 김민석

1. C++

A. 접두사 std::는 시스템 standard library에서 가져오는 것을 의미한다

B. #으로 시작하는 커맨드는 c/c++ statement가 아니라 전처리 statement다

i. File->preprocessor->compiler

C. 포인터

i. Type T에 대해서 T* Type는 T에 대한 포인터

ii. &x는 x의 메모리에서 주소를 가져오는 연산

iii. *x는 역참조로 x의 주소에 있는 값(또는 object)를 가져오는 연산

iv. 포인터 변수의 크기는 운영체제에 따라 다르다. 64비트 운영체제면 8바이트, 32비트 운영체제면 4바이트

v. 문자열 변수는 문자열의 시작점의 주소를 담고 있다

	char형 배열	char형 포인터
가리키는 위치 변경	불가능	가능
문자열 내용을 변경	가능	불가능

위 표의 특성을 실습해보도록 하겠다. 중요한 부분이니 숙지를 하기를 권장한다.

```
#include <stdio.h>

int main(void)
{
   char str1[] = "My String";
   char * str2 = "Your String";
   printf("%s %s \n", str1, str2);

   str2 = "Our String";   // char형 포인터 가르키는 대상 변경
   printf("%s %s \n", str1, str2);

   str1[0] = 'X';   // char형 배열 문자열 변경 성공
   str2[0] = 'X';   // 문자열 변경 실패
   printf("%s %s \n", str1, str2);
   return 0;
}
```

D. 문자열 char str[n]="asdfawsdf";

i. 문자열은 문자의 행렬로 문자열 변수는 행렬의 첫번째 주소인 base address를 담고 있다

ii. 문자열 변수는 처음에 할당한 주소를 바꿀 수 없다. 그러므로 pointer constant라고도 한다

ii. Char 또는 char을 이용한 문자열을 cout으로 출력할 때 &연산자로 주소를 출력하려고 해도 cout에서 자동으로 주소가 아니라 할당된 값을 출력하기 때문에 (void *)&ch와 같은 형태로 형변환을 한 후에 출력해야 주소를 알 수 있다.

v. 원래 문자열 크기는 끝을 알리는 '₩0'까지 생각해서 만들어야 한다. (4글자를 저장하는 변수는 크기가 5는 되어야 한다)

하지만 굳이 그럴 필요 없이 4글자 저장하는 문자열 크기를 4로 해도 된다. 다만 그 경우에는 문자열의 끝을 알리는 '₩0'이 나올 때까지 계속 출력할 것이다

E. 변수의 범위: 특정 범위에서 선언된 변수는 그 범위 밖에서는 쓰일 수 없다. 만약 다른 범위에 같은 변수가 선언되었다면 범위상 더 가까운 변수를 따른다

F. 글로벌 변수와 namespace: 글로벌 변수는 모든 범위에서 쓰여서 중복되거나 오용될 수 있다. 그래서 related한 것들을 모아 놓은 그룹인 namespace를 써서 구분한다.

   i. Namespace 전체를 쓸 경우에는 using namespace namespaceName;

   ii. Namespace 안의 특정한 것만 쓸 경우에는 using namespaceName::specificThing

G. Member selection operator

   i. 단순한 변수에서 member을 선택할 경우: class_name.member

   ii. 변수의 포인터에서 member을 선택할 경우: pointer->member

H. 증감연산자

   i. Post operator: 값을 리턴하고 증감을 하기 때문에 잘 안쓰임

   ii. Pre operator: 증감하고 증감한 값을 리턴한다

I. Bitwise operator

Bitwise Operators

Bitwise Operators

The following operators act on the representations of numbers as binary bit strings. They can be applied to any integer type, and the result is an integer type.

~ exp	bitwise complement
exp & exp	bitwise and
exp ^ exp	bitwise exclusive-or
exp | exp	bitwise or
exp1 << exp2	shift exp1 left by exp2 bits
exp1 >> exp2	shift exp1 right by exp2 bits

J. In-line function: 짧은 명령어로 구성된 함수이기에 원래 함수가 시스템의 call-return 메커니즘으로 움직이던 것과 달리 그 자리에서 코드를 실행한다

   i. Inline int min(int x, int y) {return(x<y?x:y)}

K. Class

i. Private member와 friend class: 원래 private member는 그 클래스에서만 접근할 수 있지만 class에 friend class를 명시하면 friend class인 클래스에서는 private member에도 접근할 수 있다

```
class Passenger {
private:
  // ...
public:
  Passenger();                          // default constructor
  Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
  Passenger(const Passenger& pass);  // copy constructor
  // ...
};
```

```
Passenger::Passenger() {                // default constructor
  name = "--NO NAME--";
  mealPref = NO_PREF;
  isFreqFlyer = false;
  freqFlyerNo = "NONE";
}
                                        // constructor given member values
Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
  name = nm;
  mealPref = mp;
  isFreqFlyer = (ffn != "NONE");        // true only if ffn given
  freqFlyerNo = ffn;
}
                                        // copy constructor
Passenger::Passenger(const Passenger& pass) {
  name = pass.name;
  mealPref = pass.mealPref;
  isFreqFlyer = pass.isFreqFlyer;
  freqFlyerNo = pass.freqFlyerNo;
}
```

```
Passenger p1;                           // default constructor
Passenger p2("John Smith", VEGETARIAN, 293145); // 2nd constructor
Passenger p3("Pocahontas", REGULAR);    // not a frequent flyer
Passenger p4(p3);                       // copied from p3
Passenger p5 = p2;                      // copied from p2
Passenger* pp1 = new Passenger;         // default constructor
Passenger* pp2 = new Passenger("Joe Blow", NO_PREF); // 2nd constr.
Passenger pa[20];                       // uses the default constructor
```

27

생성자는 여러가지(그냥 초기화, 값 넘기며 초기화, 복사 etc)로 활용할 수 있다

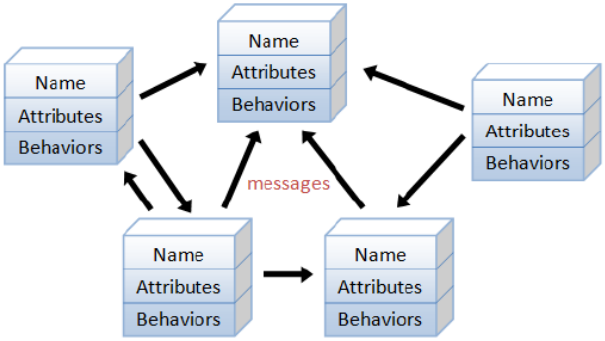
2. Object Oriented Design: 기존의 절차지향적 디자인은 software component의 재사용이 힘들었기 때문에 재사용이 쉬운 방식으로 하는 디자인

A. Class와 Object의 차이: Class는 설계도이고 Object는 설계도인 Class로 만든 물체이다

B. OOP에서 프로그램들은 Object들끼리 message를 주고받으며 작동한다

- Objects of the program interact by sending messages to each other
  - rect.setWidth(2);

object message information



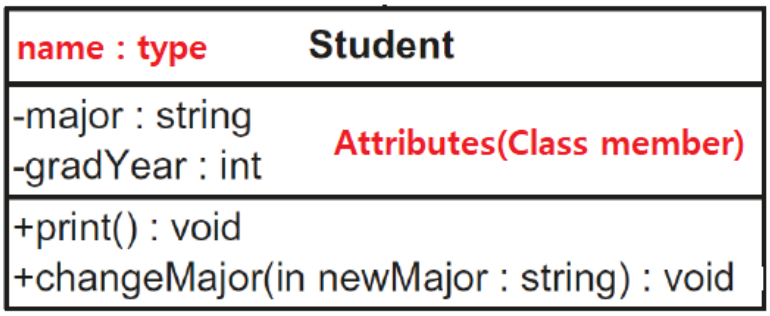
An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

C. Unified Modeling Language(UML) Class Diagram

+ is for public

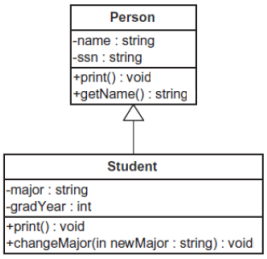
- is for private

Class name



Operations(member functions)

name(parameters : types) : return type



Student가 Person을 Inheritance(상속)하고 있다

D. 상속한 함수와 생성자 소멸자

i. 생성자: super class 생성자 -> sub class 생성자

ii. 소멸자: sub class 소멸자 -> super class 소멸자

E. Overloading과 Overriding

i. Overloading: 같은 class 내에서 같은 이름의 method, 다른 파라미터와 기능

ii. Overridding: super class와 sub class 사이에 같은 이름의 method

F. Protected: 본 class 또는 이 class를 상속하는 class에서는 public이지만 그 외에서는 private로 보이는 권한

G. Inheritance types: super class를 sub class가 어떤 타입으로 상속하냐에 따라 super class에서의 접근 권한이 달라진다

상속형태	Base Class	Inherited
public	public	public
	private	접근x
	protected	protected
private	public	private
	private	접근x
	protected	private
protected	public	protected
	private	접근x
	protected	protected

```cpp
#include <iostream>

using namespace std;

class Base {
private:
        int pri;
public:
        int pu;
        Base(){
                pri = 1;
                pu = 2;
                pro = 3;
        }
protected:
        int pro;
};


class Derived1 : public Base {
public:
        void print() {
                //cout << pri << endl;
                cout << pro << endl;
                cout << pu << endl;
        }
};

class Derived2 : protected Base {
public:
        void print() {
                //cout << pri << endl;
                cout << pro << endl;
                cout << pu << endl;
        }
};


int main() {

        Derived1 d1;
        Derived2 d2;

        d1.print();
        d2.print();

        cout << d1.pu << endl;
        //cout << d2.pu << endl;
        return 0;
}
```

```
hjnam@CS:~/examples$ ./02_inheritance_derive.out
3
2
3
2
2
hjnam@CS:~/examples$
```

주석 처리한 cout<<d2.pu<<endl;은 Derived2에서 protected로 상속받아서 모든 변수가 protected가 되었기에 main함수에서 접근 할 수 없고 위의 d2.print()는 Drived2 내에서 접근하는거라 가능함

H. Binding: 함수 호출을 실제 함수의 몸체와 연결하는 것

i. Static binding: 변수 선언할 때 무슨 타입인지는 상관없이 포인터 타입에 따라서 컴파일 도중에 binding을 한다

1. 장점: 미리 타입을 정해놓기 때문에 안정적이고 dynamic binding에 비해 리소스를 아낄 수 있다

2. 단점: 타입을 바꿀 수 없어 flexible하지 않다

```
Person* pp[100];                    // array of 100 Person pointers
pp[0] = new Person(...);            // add a Person (details omitted)
pp[1] = new Student(...);           // add a Student (details omitted)
```

```
cout << pp[1]->getName() << '\n';  // okay
pp[0]->print();                    // calls Person::print()
pp[1]->print();                    // also calls Person::print() (!)
pp[1]->changeMajor("English");     // ERROR!
```

The reason for this apparently anomalous behavior is called ***static binding***—when determining which member function to call, C++'s default action is to consider an object's ***declared type***, not its actual type. Since *pp*[1] is declared to be a pointer to a Person, the members for that class are used. Nonetheless, C++ provides a way to achieve the desired dynamic effect using the technique we describe next.

처음 Person 포인터에 따라서 컴파일 중에 바인딩했기에 pp[1]->changeMajor();은 Person에서 메소드를 찾지만 없기 때문에 Error가 생긴다.

ii. Dynamic binding: 컴파일 중에는 binding을 미루고 실제 런타임 중에 변수 타입을 보고 binding을 한다

1. Dynamic binding할 함수는 virtual 키워드 붙여야 한다
2. 장점: 다형성을 실현할 수 있고 flexible하다
3. 단점: 안정성이 떨어지고 리소스가 많이 든다
4. 자세한 부분은 여기 참조
   A. https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=dd1587&logNo=221119668089
   B. https://jerryjerryjerry.tistory.com/13

- The decision as to which function to call is made at run-time, hence the name ***dynamic binding***.

declare the print function to be virtual.

```
class Person {                          // Person (base class)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
class Student : public Person {         // Student (derived from Person)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
```

Let us consider the effect of this change on our array example, thereby illustrating the usefulness of dynamic binding.

```
Person* pp[100];                        // array of 100 Person pointers
pp[0] = new Person(...);                // add a Person (details omitted)
pp[1] = new Student(...);               // add a Student (details omitted)
pp[0]->print();                         // calls Person::print()
pp[1]->print();                         // calls Student::print()
```

Print()에 virtual을 붙였기에 pp[1]->print()에서도 student의 print()를 호출한다

iii. Static binding과 Dynamic binding의 차이: 컴파일 중 할당 되는 것은 stack 메모리에 할당되고 런타임 중에 할당 되는 것은 heap 메모리에 할당된다

1. Stack/heap과 정적 할당 동적 할당은 여기를 참조해라

A. https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=eludien&logNo=221462352935

I. Polymorphism(다형성): 적어도 하나의 virtual 함수를 가지고 있는 class object를 가르키는 포인터 변소를 polymorphic하다고 한다

J. Interface/Abstract Class: 상속할 base class로만 쓰기 위한 class이다

i. Object를 생성할 수 없다

ii. C++에서는 하나 이상의 function이 abstract(pure virtual)하면 abstract class이다

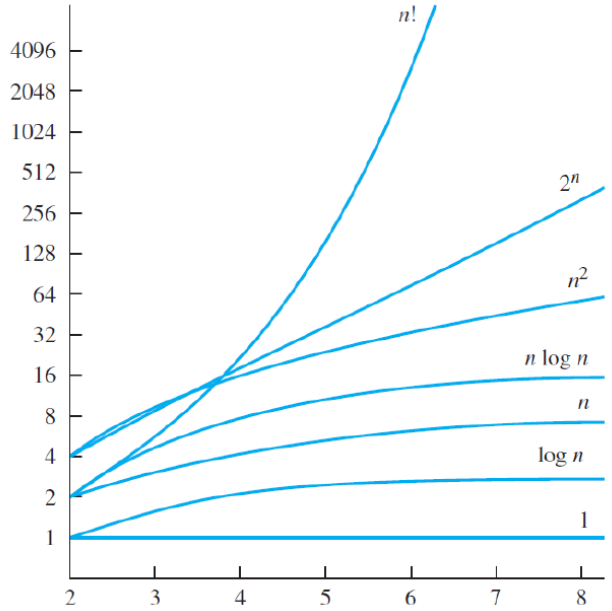
1. abstract(pure virtual)함은 function body 부분을 =0으로 표기할 때 만들 수 있다

K. Template: 임의의 타입 T에 대한 generic function/class을 만들기 위한 기능

i. Function/class 앞에 template <typename T>를 붙여서 만든다

3. Analysis

A. Input size가 증가할수록 증가하며 하드웨어/소프트웨어 환경에 독립적인 성능을 재는 것을 목표로 하며 guarantee를 위해 보통 worst case를 고려한다



B. 알고리즘에 의해 계산되는 basic computation 수로 결정되며 pseudocode로도 알 수 있다. 또한 프로그래밍 언어에 독립적이다

4. ArrayList(배열): array 개념을 object까지 저장할 수 있는걸로 확장시킨 것

   A. element들은 index(앞에 선행하는 object의 수)로 접근, 삽입, 삭제가 가능하다

   B. UML

ArrayList
-MaxEntries : int(array 최대 개수 저장) -numEntries : int(현재 array에 저장된 element 갯수) -Array* : array(element를 저장할 array)
+ArrayList(argv) : void(생성자) +~ArrayList() : void(소멸자) +at(index : int) : object(index에 있는 element 리턴) +set(index : int, obj : object) : void(index에 있는 element를 obj로 대체) +insert(index : int, obj : object) : void(index에 obj를 삽입) +erase(index : int) : object(index에 있는 element 삭제하고 삭제한 값 리턴) +size() : int(현재 array안에 있는 element 수 리턴) +empty() : bool(현재 array가 비어있는지 리턴)

대체적인 ArrayList UML

   i. at~erase는 index가 범위를 벗어났을 경우 exception dealing해야함

   C. Insert, Erase: i번째에 element를 삽입/삭제하고 뒤의 element들을 다 조정해준다

   D. Performance

      i. size, empty, at, set은 O(1)로 작동하지만 insert, erase는 뒤에 것들을 다 조정해야 되기 때문에 O(n)으로 작동한다

         1. 그러므로 자신이 어떤 것을 구현할 것인가를 따져서 자료구조를 정하자!

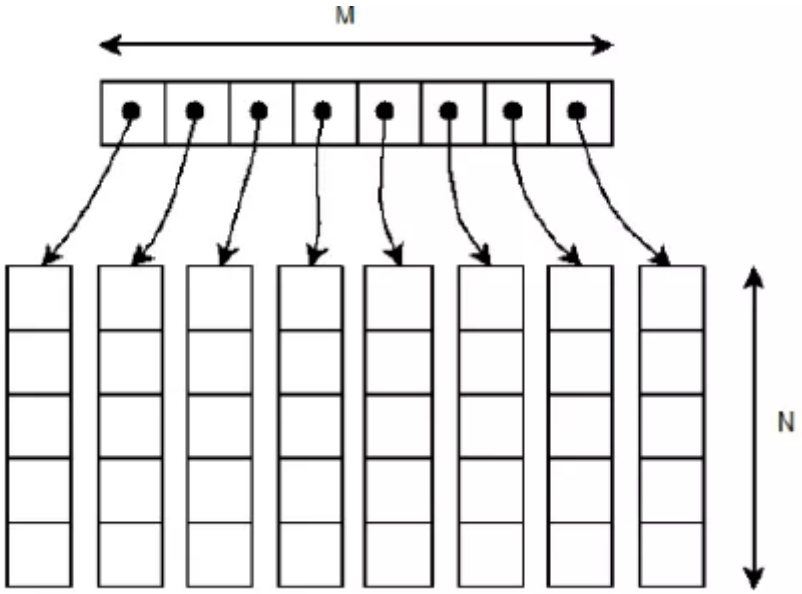
   E. Array가 full인데 insert를 쓸 경우

      i. 다 찼으니 안된다고 error handling을 한다

      ii. 원래 크기의 2배인 array를 생성하고 거기에 원 array의 element들을 복사한다

   F. Insert, set등 할 때 주의할 점: pass by reference를 위해 object& varName 형태로 하자

G. Matrix(two-dimensional arraylist): row를 나타내는 arraylist를 여러 개 만들어 한번에 운용

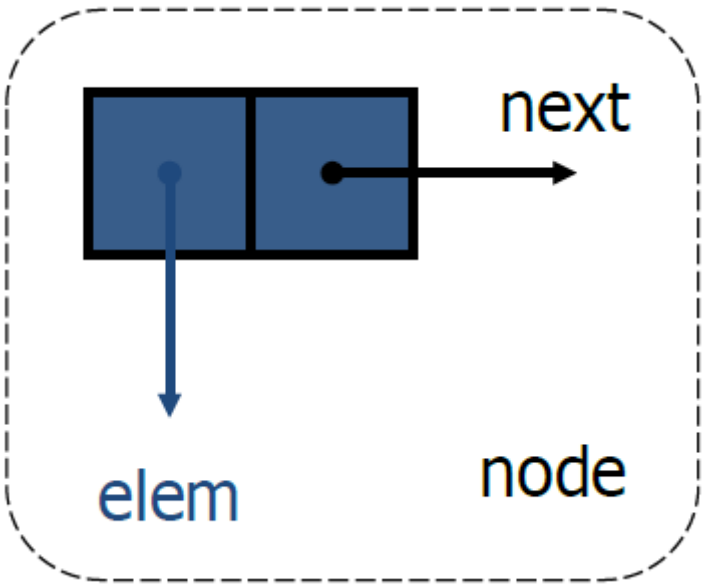


i. 생성: 각 row에 대한 포인터 arraylist를 만들고 각 row를 생성하는 arraylist를 생성한다

ii. 소멸: 각 row를 나타내는 arraylist를 없애고 row에 대한 포인터 arraylist를 없앤다

5. LinkedList

A. Singly Linked List: element와 다음 노드를 가리키는 link로 이루어진 node의 sequence로 이루어진 자료구조



i. Insert: 새 node가 들어갈 위치에 있던 node와 link를 만들고 그 node에 연결되 있던 노드와 삽입할 노드 간에 link를 만든다

ii. Remove: 제거할 node에 원래 연결되있던 link들을 제거할 node 가리키던 곳에 연결하고 node를 삭제한다

iii. Insert/remove를 head/tail에서 할 때의 차이: tail에서 삭제하려면 head에서 끝까지 이동해야 하기 때문에(O(n)의 시간 복잡도) head에서 삭제하는걸로 만든다

iv. 시작과 끝은 SLL의 head와 NULL로 지정한다

v. UML

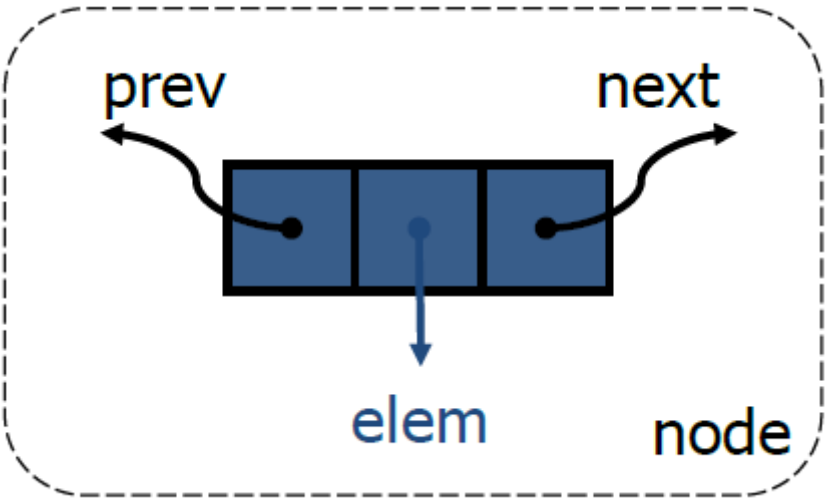
SinglyLinkedList
-header : TNode*(head로 기능할 Node의 포인터)
+SinglyLinkedList() : void(생성자) +~SinglyLinkedList() : void(소멸자) +empty() : bool(현재 SinglyLinkedList가 비어있는지 리턴) +front() : T&(현재 가장 앞의 elemen를 리턴) +addFront(T& e) : void(가장 앞에 element e를 가진 node 삽입) +removeFront() : void(가장 앞의 node 삭제)

Front에서 insert/remove하는게 효율이 좋아서 front에서 하는 것만 구현

vi. ArrayList vs LinkedList

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

B. Doubly Linked List: 이전/다음 Node 포인터, element로 이루어진 node의 sequence로 이루어진 자료구조



i. 시작과 끝은 DLL의 header와 trailer로 정한다

ii. SLL vs DLL

1. SLL은 앞 Node의 Link로만 접근 가능한 반면에 DLL은 앞 뒤의 Link로 둘 다 접근 가능하다
2. SLL은 한 Node 당 2 field만 쓰지만 DLL은 한 Node 당 3 field를 쓴다
3. DLL이 SLL보다 접근 같은 여러 기능을 쓰기 쉽다

iii. 모든 연산이 O(1)이다

iv. UML

DLinkedList
-header : DNode*(head로 기능할 Node의 포인터) -trailer : DNode*(tail로 기능할 Node의 포인터)
+DLinkedList() : void(생성자) +~DLinkedList() : void(소멸자) +empty() : bool(현재 DLinkedList가 비어있는지 리턴) +front() : T&(현재 가장 앞의 element를 리턴) +back() : T&(현재 가장 뒤의 element를 리턴) +addFront(T& e) : void(가장 앞에 element e를 가진 node 삽입) +addBack(T& e) : void(가장 뒤에 element e를 가진 node 삽입) +removeFront() : void(가장 앞의 node 삭제) +removeBack() : void(가장 뒤의 node 삭제) Protected –add(DNode* v, T& e) : void(v위치에 e를 element로 가지는 Node 삽입) Protected –remove(DNode* v) : void(v위치에 있는 Node 삭제)

1. Protected 함수들은 public함수에서 쓰일 것들이다

C. Circularly Linked List: 순환적으로 이어줘서 head와 tail이 없는 SLL

i. UML

CircleList
-cursor : CNode*(현재 가리키는 Node의 포인터)
+CircleList() : void(생성자) +~CircleList() : void(소멸자) +empty() : bool(현재 CircleList가 비어있는지 리턴) +front() : T&(cursor Node에서 reference하고 있는 Node의 element) +back() : T&(cursor가 가리키는 Node의 element) +advance() : void(cursor를 현재 cursor Node의 다음 Node로 이동시킨다) +add(T& e) : void(cursor Node 다음에 e를 element로 가지는 Node 삽입) +remove() : void(cursor Node 다음에 있는 Node 삭제)

6. Stack: FILO(LIFO)를 원칙으로 물건을 차곡차곡 쌓듯이 보관하는 자료구조

A. UML

LinkedStack
-S : SLL<T>(element들을 저장할 LL) -n : int(# of elements)
+LinkedStack() : void(생성자) +~LinkedStack() : void(소멸자) +size() : int(LinkedStack 내의 element 수 리턴) +empty() : bool(현재 LinkedStack가 비어있는지 리턴) +top() : T&(가장 위의 element 리턴) +push(T& e) : void(LinkedStack의 가장 위에 element e를 추가한다) +pop() : void(LinkedStack의 가장 위에서 element를 하나 삭제한다)

Pop, top은 StackEmpty를 push는 StackFull을 dealing해줘야 함

B. 공간 복잡도는 O(n), 시간 복잡도는 O(1)

7. Queue: FIFO(LILO)을 원칙으로 파이프를 지나는 물처럼 보관하는 자료구조

A. CircleList-Based queue

i. UML

LinkedQueue
-C : CircleList<T>(element들을 저장할 CircleList) -n : int(# of elements)
+LinkedQueue() : void(생성자) +~LinkedQueue() : void(소멸자) +size() : int(LinkedQueue 내의 element 수 리턴) +empty() : bool(현재 LinkedQueue가 비어있는지 리턴) +front() : T&(가장 앞의 element 리턴) +enqueue(T& e) : void(LinkedQueue의 가장 뒤에 element e를 추가한다) +dequeue() : void(LinkedQueue의 가장 위에서 element를 하나 삭제한다)

ii. Enqueue는 QueueFull을 front, dequeue는 QueueEmpty를 dealing해줘야 한다

B. Simple Array-Based Queue

i. 넣는 것은 뒤에서 빼는 것은 앞에서부터 해야되기 때문에 앞과 뒤를 둘 다 기억해야 O(1)로 실행할 수 있다

ii. Queue최대 크기를 N, 현재 element 수를 n, 가장 앞의 element 위치 index를 f, 가장 뒤의 element 바로 뒤의 element 위치 index를 r로 하여 저장된 공간을 넘어가지 않게 index+1이 N을 넘어가면 mod N 연산을 하여 N보다 작게 하여 앞과 뒤의 위치를 보관하여 바로 접근하도록 한다

C. Double-Ended Queue(Deque: 읽을 때는 dequeue랑 헷갈리지 않게 deck이라 읽는다): DLL을 활용하여 앞과 뒤에서 삽입/삭제가 가능하도록 하는 자료구조

i. UML

LinkedDeQue
-D : DLinkedList<T>(element들을 저장할 DLL) -n : int(# of elements)
+LinkedDeQue() : void(생성자) +~LinkedDeQue() : void(소멸자) +size() : int(LinkedDeQue 내의 element 수 리턴) +empty() : bool(현재 LinkedDeQue가 비어있는지 리턴) +front() : T&(가장 앞의 element 리턴) +back() : T&(가장 뒤의 element 리턴) +insertFront(T& e) : void(LinkedDeQue의 가장 앞에 element e를 추가한다) +insertBack(T& e) : void(LinkedDeQue의 가장 뒤에 element e를 추가한다) +removeFront() : void(LinkedDeQue의 가장 앞에서 element를 하나 삭제한다) +removeBack() : void(LinkedDeQue의 가장 뒤에서 element를 하나 삭제한다)

8. Tree

A. Tree traversal

i. Preorder: 특정 node를 방문한 다음에 descendant node들을 방문한다

```
Algorithm preorder(v)
    visit(v)
    for each child w of v
        preorder (w)
```

ii. Postorder: 특정 node의 descendant node들을 방문한 다음에 특정 node를 방문한다

```
Algorithm preorder(v)
    visit(v)
    for each child w of v
        preorder (w)
```

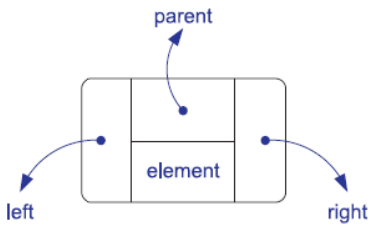
iii. Inorder: 특정 node의 left subtree>특정 node>특정 node의 right subtree순으로 방문한다

```
Algorithm inOrder(v)
    if ¬ v.isExternal()
        inOrder(v.left())
    visit(v)
    if ¬ v.isExternal()
        inOrder(v.right())
```

B. Binary Tree

i. Property

1. 모든 internal node는 최대 2 children을 가진다(모든 internode가 2 children을 가지면 proper binary tree라고 한다)
2. Internal node의 children들은 left child, right child로 나뉜다
3. Proper binary tree: 모든 internal node가 2 children을 가질 때
   A. Property: n- # of nodes, e- # of external nodes, i- # of internal nodes, h- height
      i. e=i+1
      ii. n=e+i=2e-1=2i+1
      iii. h<=i
      iv. e<=2^h
      v. h>=log2(e)
   B. binary tree의 node: element, parent node, left child node, right child node에 대한 포인터로 구성되어 있다



C. UML

i. LinkedBinaryTree

LinkedBinaryTree
-_root : Node*(root node의 포인터를 저장함) -n : int(# of nodes)
Protected –Node : class def(Node class에 대한 definition) +Position : class def(Position class에 대한 definition) +PositionList : std::list<Position>(PositionList list에 대한 typedef) +LinkedBinaryTree() : void(생성자) +~LinkedBinaryTree() : void(소멸자) +size() : int(LinkedBinaryTree 내의 element 수 리턴) +empty() : bool(현재 LinkedBinaryTree가 비어있는지 리턴) +root() : Position(root의 Position을 리턴한다) +positions() : PositionList(모든 node들의 Position을 리턴한다) +addRoot() : void(빈 tree에 root node를 추가한다) +expandExternal(p : Position&) : void(p에 존재하는 node에 external node들을 추가하여 tree를 확장한다) +removeAboveExternal(p : Position&) : Position(p와 p의 parent Node를 삭제한다) Protected –preorder(v : Node*, pl : PositionList&)(preorder로 tree탐색)

ii. Position

Position
-v : Node*(Position이 가리킬 node의 포인터)
+Position()(생성자) +~Position()(소멸자) +operator*() : T&(Position이 가리키는 node의 element를 리턴한다) +left() : Position(node의 left를 가리키는 Position을 리턴한다) +right() : Position(node의 right를 가리키는 Position을 리턴한다) +parent() : Position(node의 par를 가리키는 Position을 리턴한다) +isRoot() : bool(node가 root node인지 알려준다) +isExternal() : bool(node가 external node인지 알려준다) Friend class LinkedBinaryTree(LinkedBinaryTree에서 접근할 수 있도록 한다)

9. PriorityQueue

A. Property

i. 넣은 순서와 상관없이 priority가 낮은 순서대로 나오는 자료구조

ii. Priority를 비교해야 하기 때문에 보통 element가 (key, value)형태이며 key로 priority를 비교한다

B. 구현 방식 차이: 삽입할 때 sorting vs 제거할 때 sorting

i. 삽입할 때 sorting(insertion-sort)

1. 삽입할 때 O(n^2), 제거할 때 O(n)이며 읽기가 많은 경우에 적합

ii. 제거할 때 sorting(selection-sort)

1. 삽입할 때 O(n), 제거할 때 O(n^2)이며 쓰기가 많은 경우에 적합

C. UML

ListPriorityQueue template <E, C>
-L : std::list<E>(type E의 element들이 저장될 list) -isLess : C(priority를 비교할 비교식이 들어 있는 type C의 변수)
+ListPriorityQueue()(생성자) +~ListPriorityQueue()(소멸자) +size() : int(# of elements) +empty() : bool(ListPriorityQueue가 비었는지 리턴) +insert(E& e) : void(ListPriorityQueue에 e라는 element를 삽입) +min() : E&(priority가 가장 낮은 element를 리턴) +removeMin() : void(priority가 가장 낮은 element를 삭제)

10. Heap

A. Property

i. Complete binary tree: level i(depth i)에서 2^i개의 node를 가지고 있으며 node들은 왼쪽에서 오른쪽으로 채워진다

ii. Heap-Order: root node를 제외한 모든 node v는 key(v)>=key(parent(v))를 만족한다

iii. 마지막 node는 가장 큰 depth를 가지고 rightmost하다

B. Insertion and Upheap & removal and Downheap

i. Insertion and Upheap

1. Insertion: 새로운 last node z에 새로운 element(k, v)를 저장한다
2. Upheap: parent node와 priority를 비교하여 heap-order가 맞을 때까지 swapping한다
3. Node가 n개인 heap의 height는 floor(logn)이기 때문에 upheap은 O(logn)의 시간복잡도를 가진다

ii. Removal and Downheap

1. Removal: root node와 last node를 바꾼 후 바뀐 last node를 없앤다
2. Downheap: 바뀐 root node의 priority를 child node(뭐랑 바꾸는지는 구현자 마음)와 priority를 비교하여 heap-order가 맞을 때까지 swapping한다
3. Upheap랑 같은 이유로 O(logn)의 시간복잡도를 가진다

C. Heap-sort

i. Heap의 공간복잡도가 O(n), insert,removal 시간복잡도가 O(logn)이기 때문에 heap-sort의 시간복잡도는 O(nlogn)이며 insertion-sort, selection-sort보다 효율이 좋다

D. UML

i. Vector based-Heap implementation

1. Property

A. index관련 연산을 편하게 하기 위하여 n개의 element를 담을 때 n+1개의 vector을 사용하며 index=0는 쓰지 않는다

B. index 관련 property: for node at index i

i. 만약 node가 root node라면 index=1

ii. Left child의 index는 2i

iii. Right child의 index는 2i+1

C. LinkedList가 아니라 Array이니까 link는 없다

2. VectorCompleteTree

VectorCompleteTree template <E>
-V : std::vector<E>(type E의 element들이 저장될 vector) +Position : std::vector<E>::iterator typedef typename(vector에 접근할 iterator 가독성 높이기 위한 typedef)
+VectorCompleteTree()(생성자) +~VectorCompleteTree()(소멸자) +size() : int(# of elements) +pos(i : int) : Position(index에 맞는 position을 리턴한다) +idx(p : Position&) : int(Position에 맞는 index를 리턴한다) +left(p : Position&) : Position(left child의 Position을 리턴한다) +right(p : Position&) : Position(right child의 Position을 리턴한다) +parent(p : Position&) : Position(parent의 Position을 리턴한다) +hasLeft(p : Position&) : bool(left child의 존재여부를 리턴한다) +hasRight(p : Position&) : bool(Right child의 존재여부를 리턴한다) +isRoot(p : Position&) : bool(root node인지의 여부를 리턴한다) +root() : Position(root node의 Position을 리턴한다) +last() : Position(last node의 Position을 리턴한다) +addLast(e : E&) : void(e라는 값을 가지는 element를 마지막에 추가한다) +removeLast() : void(last node를 삭제한다) +swap(p : Position&, q : Position&) : void(p와 q의 element를 바꾼다)

3. HeapPriorityQueue

HeapPriorityQueue template <E, C>
-T : VectorCompleteTree<E>(type E의 element들이 저장될 vectorcompletetree) -isLess : C(priority를 비교하기 위한 연산자)
+HeapPriorityQueue()(생성자) +~HeapPriorityQueue()(소멸자) +size() : int(# of elements) +empty() : bool(HPQ가 비었는지 여부를 리턴한다) +insert(e : E&) : void(e를 Element로 가지는 node를 삽입한다) +min() : E&(priority가 가장 낮은 element을 리턴한다) +removeMin() : void(priority가 가장 낮은 node를 삭제한다)

11. Map: (key, value) element들을 가지고 key를 통해서 search, insert, remove하기 때문에 같은 key를 가진 여러 element들은 허용되지 않는다

A. UML

i. Entry

Entry template <K, V>
-_key : K(Entry의 key) -_value : V(Entry의 value)
+Entry(k : K&, v : V&)(생성자) +~Entry()(소멸자) +key() : K&(Entry의 _key를 리턴한다) +value() : V&(Entry의 _value를 리턴한다) setKey(k : K&) : void(_key를 k로 set한다) setValue(v : V&) : void(_value를 v로 set한다)

ii. Map

Map template <K, V>
+Entry : class(Map의 Entry class) +Iterator : class(Entry에 접근하기 쉽게 해주는 iterator class)
+Map()(생성자) +~Map()(소멸자) +size() : int(# of elements) +empty() : bool(Map이 비어있는지 여부 리턴한다) +find(k : K&) : Iterator(key k를 가지는 element를 찾아 iterator 리턴한다) +put(k : K&, b : V&) : Iterator((k, v)인 element를 새로 삽입하거나 replace한 후 iterator 리턴한다) +erase(k : K&) : void(key k를 가진 element를 삭제한다) +erase(p : Iterator&) : void(Iterator p 자리에 있는 elemen를 삭제한다) +begin() : Iterator(시작점의 Iterator을 리턴한다) +end() : Iterator(끝점의 Iterator을 리턴한다)

erase(k : K&)는 key가 Map에 없을 경우를 상정하여 NonexistentElement를 dealing해야 한다

B. A Simple List Based Map

i. Put, find, erase 모두 link로 처음부터 끝까지 가야하기 때문에 시간복잡도O(n)을 가져서 작은 size의 map이 아니면 효율이 떨어진다

C. HashMap: hashing function을 거친 key값을 index로 해서 그 index에 element를 저장하는 Map

i. Bucket Array: Array의 각 칸마다 bucket을 달아서 그 bucket에 element들을 보관한다

ii. Hash function

1. Hash code: compression function에 사용하기 위한 알맞은 형태로 변환해주는 기능(e.g., keys->integers)

2. Compression function: integer들을 특정 범위로 한정시켜 Map의 범위 내 int로 index를 바꿔준다

iii. UML

1. Iterator

Iterator
-ba : BktArray*(bucketarray의 Iterator) -bkt : BItor(Bucket용 Iterator) -ent : Eitor(Entry용 Iterator)
+Iterator(a : BktArray&, b : BItor&, c : EItor)(생성자) +~Iterator()(소멸자) +operator*(): Entry&(*연산자 오버로딩) +operator==(p : Iterator&) : bool(==연산자 오버로딩) +operator!=(p : Iterator&) : bool(!=연산자 오버로딩) +operator++() : Iterator&(++연산자 오버로딩) Friend class HashMap

2. HashMap

HashMap template<K, V, H>
-n : int(# of elements) -hash : H(hashing function) -B : BktArray(Bucket들을 저장할 BktArray) +Entry : class(Entry class definition) +Iterator : class(Iterator class definition) Protected –Bucket : list<Entry>(Bucket typedef) Protected –BktArray : vector<Bucket>(BktArray typedef) Protected –BItor : BktArray::Iterator(BItor typedef) Protected –EItor : Bucket::Iterator(EItor typedef)
+HashMap(capacity : int)(생성자) +~HashMap()(소멸자) +size() : int(# of elements) +find(k : K&) : Iterator(finder 함수의 container) +put(k : K&, v : V&) : Iterator(inserter 함수의 container) +erase(k : K&) : void(eraser 함수의 container) +erase(p : Iterator&) : void(eraser 함수의 container) +begin() : Iterator(BktArray의 첫부분을 리턴) +end() : Iterator(BktArray의 끝부분을 리턴) Protected –finder(k : K&) : Iterator(k를 key로 가지는 것을 찾아 리턴) Protected –inserter(p : Iterator&, e : Entry&) : Iterator(p 위치에 e를 삽입) Protected –eraser(p : Iterator&) : void(p 위치에 있는 것을 삭제) Protected –nextEntry(p : Iterator&) : void(Iterator를 다음 entry로 이동시키는 함수)

erase(k : K&)는 key가 HashMap에 없을 경우 keyException을 dealing해야 한다

12. SearchTree(안함 귀찮음 ㅅㄱ)