**Binomial Heap 구현 및 실험**

**1. 서론**

Binomial Heap을 C로 구현하고 C++에서 제공하는 binary heap 알고리즘과 비교 실험을 진행하였다.

**2. 구현**

구현은 Introduction to Algorithm의 Binomial Heap 파트를 참고했다.

**2.1. bheap.{c,h}**

**#define TRUE (1==1)**

**#define FALSE (!TRUE)**

**#define SUCCESS (0)**

**#define FAILURE (1)**

에러 처리와 예외 값 처리를 위한 상수다.

**typedef int Key;**

원하는 데이터를 Key로 설정하여 사용할 수있다. 이번 실험에서 Key는 int로 설정하였다.

**typedef struct Node {**

**Key key;**

**int degree;**

**struct Node\* parent;**

**struct Node\* child;**

**struct Node\* sibling;**

**} Node;**

Binomial Heap을 구성하는 노드를 정의하는 구조체이다.

**// Empty heap is NULL pointer(0 Node).**

빈 힙은 노드가 하나도 없는 힙으로, Node\* 변수에 NULL을 할당하여 생성한다.

**#define GET(x, prop, none) ((x) ? (x)->prop : (none))**

**void print\_node(Node\* node);**

**void print\_tree(Node\* root);**

**void print\_roots(Node\* head, Node\* tail);**

**void print\_heap(Node\* heap);**

디버깅을 위해 정의한 시각화 함수이다. 각각 노드, binomial tree, root 리스트, heap을 출력한다.

**int link\_tree(Node\* parent, Node\* child);**

두개의 노드를 부모 - 자식 관계로 연결한다. merge\_heap에서 사용한다.

**Node\* merge\_roots(Node\* heap1, Node\* heap2);**

binomial heap 둘을 merge할 때, 그들의 루트 리스트를 O(n) 시간 복잡도로 합치기 위한 함수. 합쳐진 루트 리스트의 노드는 노드의 degree에 오름차순으로 정렬된다. 공간 복잡도는 O(1)이다.

**int merge\_heap(Node\* heap1, Node\* heap2, Node\*\* merged);**

binomial heap 둘을 merge한다. merge된 힙을 가리키는 포인터를 \*merged에 할당한다. Introduction to Algorithm에 제시된 알고리즘을 충실하게 구현하였다. 시간복잡도는 두 힙에 존재하는 노드의 합 N에 대해, 최악의 경우 O(log N)이다.

**int insert(Node\*\* heap, Node\* node);**

heap에 node를 삽입한다. node를 원소가 하나 있는 heap으로 간주하고 merge\_heap을 적용한다.

Key **min**(Node\* heap, Node\*\* min\_node);

heap에서 최소 원소를 찾아 key를 반환한다. Binomial heap은 min heap 속성을 따르므로, 모든 최소 원소는 root 리스트에 존재한다. Root 리스트에서 최소 원소를 찾아서 그 key를 반환한다.

Key **pop\_min**(Node\*\* heap);

heap에서 최소 원소를 찾아 제거하고 key를 반환한다. Binomial heap은 min heap 속성을 따르므로, 모든 최소 원소는 root 리스트에 존재한다. Root 리스트에서 최소 원소를 찾아서 삭제하고, 삭제한 root의 자식들로 degree의 오름차순으로 정리하여 새로운 heap을 만든다. 이를 기존 heap과 merge한다.

**int** **num\_tree\_node**(Node\* root);

**int** **is\_heap**(Node\* heap);

heap이 binomial heap이면 TRUE를 반환, 그렇지 않으면 FALSE를 반환한다. 1) heap에 존재하는 모든 binomial tree를 구성하는 노드 수가 2의 거듭제곱인지 확인한다. 2) heap에 존재하는 모든 binomial tree가 min heap property(부모 노드의 key < 자식 노드의 key)를 만족하는지 확인한다.

**2.2. unit test: test.cpp**

Binomial heap 구현을 위해 다양한 경우를 테스트하였다.

**TEST(link\_tree, test)**

**TEST(print\_tree, test)**

보조 함수 테스트

**TEST(merge\_roots, 1)**

**TEST(merge\_roots, 2)**

**TEST(merge\_roots, nil\_h1)**

**TEST(merge\_roots, property\_test)**

루트 리스트 머지 함수 테스트. property\_test의 경우 무작위 루트 리스트를 만들고 합친 뒤 정렬되었는지 확인.

**TEST(merge\_heap, merge\_2\_empty\_heap\_return\_empty\_heap)**

**TEST(merge\_heap, merge\_empty\_and\_1node\_return\_1node)**

빈 힙과 합치는 경우 테스트

**TEST(merge\_heap, case1\_just\_link\_roots)**

**TEST(merge\_heap, case2\_included)**

**TEST(merge\_heap, case3\_link\_two\_deg2\_heap\_to\_deg4\_heap)**

**TEST(merge\_heap, case4\_link\_two\_deg2\_heap\_to\_deg4\_heap)**

Introducton to Algorithm에 제시된 4가지 케이스에 대한 테스트

**TEST(heap, insert)**

**TEST(heap, pop\_min)**

**TEST(heap, min)**

**TEST(heap, prop\_test)**

각 operation에 대한 테스트. prop\_test는 무작위로 heap을 만들고 무작위로 opertion을 수행해서 heap이 is\_heap을 통과하는지 확인한다.

**3. 실험**

**3.1. 실험 구성**

std::queue와 성능을 비교하기 위해 두가지 실험을 설계했다. N개의 아이템(1~N)에 대해

1) 1 ~ N을 모두 삽입한 후에 모두 삭제하는 경우

2) 1 ~ N을 데이터 삭제와 삽입을 50% 확률로 무작위로 수행할 경우

cbuf와 std::queue에서 걸리는 시간을 측정하였다.

실험은 n = 100번 반복 수행하였다. i번째 실험에서 N = (229 / n) \* i으로 설정하여 반복하여 실행에 걸린 시간을 측정하였다. main.cpp에서 실험을 하고,실험 결과를 yaml 포맷으로 stdout에 출력하였다. 출력된 결과를 파이썬 스크립트 plot.py를 이용하여 그래프로 나타내었다.

**4. 실험 결과**

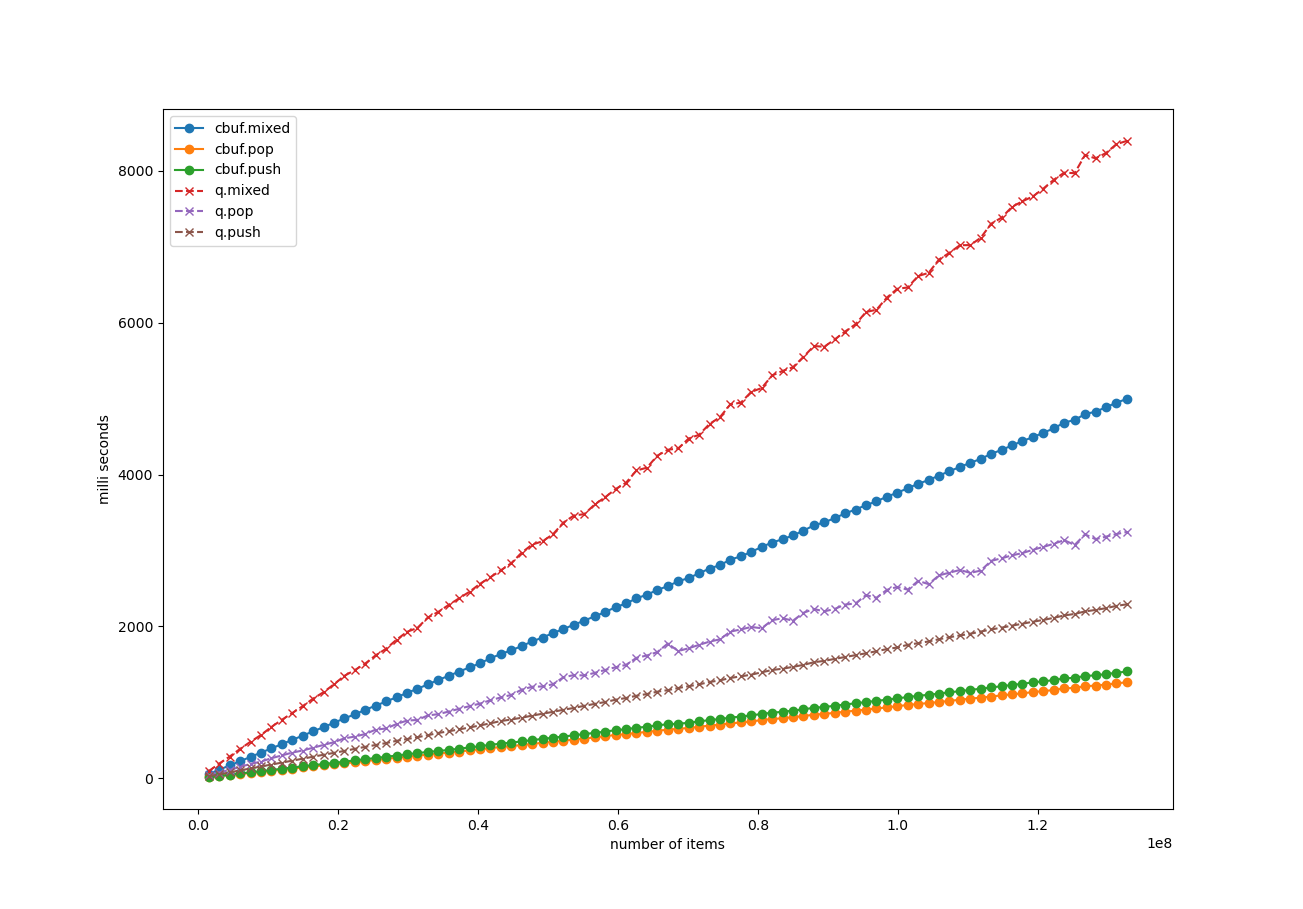


그림 1. 데이터에 대한 circular buffer, std::queue의 수행시간

cbuf가 붙는 결과는 circular buffer를 사용한 결과, q가 붙은 결과는 std::queue를 사용한 결과이다. pop, push는 3.1의 실험1)에서 각각 아이템을 삽입, 삭제하는데 걸린 시간이다. mixed는 3.1의 실험2) pop,push 혼합 작업을 수행하는데 걸린 시간이다.

표 1. 수행시간 상승 비율. 수행시간을 외삽하여 기울기를 측정하였다.

|  |  |  |
| --- | --- | --- |
| 데이터  자료구조 | queue | cbuf |
| mixed | 6.410 \* 10-5 | **3.768 \* 10-5** |
| push | 1.727 \* 10-5 | **1.056 \* 10-5** |
| pop | 2.495 \* 10-5 | **9.509 \* 10-6** |

수행 시간은 std::queue와 circular buffer 모두 데이터수 N에 비례하는 선형으로 증가하며, 전체적으로 queue보다 circular buffer의 성능이 더 낫다. queue의 경우 연결리스트로 구현되기 때문에, circular buffer와 시간 복잡도는 동일하나 실제 수행 시간 성능에서 큰 차이를 보인다. 이에 비해 circular buffer는 연속적인 배열을 힙에 선언하는 방식이기 때문에 메모리 참조의 locality가 높아 캐시를 매우 효율적으로 활용할 수 있다. 이번에 구현한 circular buffer의 단점은 삽입하려는 데이터의 수가 버퍼 사이즈보다 클 경우 정보를 저장할 수 없다는 점이다.

**5. 결론**

이번 과제에서는 circular buffer를 구현하고, std::queue와의 성능을 비교해 보았다. 시간 복잡도는 queue와 circular buffer가 동일하나, 실제 수행 시간에 대해 연속된 배열을 이용하는 circular buffer가 더 나은 성능을 보였다.