자료구조 (Data Structure)
4주차: 큐

일차원(선형) 자료구조

- 리스트: 모든 데이터를 순서로 직접 접근할 수 있다
- 스택: 가장 최근(top)의 데이터만 접근할 수 있다
- 큐: 가장 오래된(front)의 데이터만 접근할 수 있다
- 리스트는 자유 접근, 스택/큐는 제한된 접근이지만 효율적

큐

- 특징: First-In First-Out (FIFO) 구조
- 정의: 한쪽 끝(rear)에서만 추가, 다른 쪽 끝(front)에서만 삭제가 가능한 선형 자료구조
- 기능:
  - ENQUEUE(x): rear에 데이터 x를 추가
  - DEQUEUE(): front에서 데이터 삭제/반환
- 시간 효율: ENQUEUE / DEQUEUE 모두 O(1)

단방향 링크드 리스트로 구현한 큐

- 특징: 노드는 뒤(rear)에 추가하기 쉽고, 앞(front)에서 삭제하기 쉽다

- front: 삭제될 첫 노드 (초기값 NULL)
- rear: 가장 나중에 추가된 마지막 노드 (초기값 NULL)

- LQUEUE_ENQUEUE(q, value):
  새 노드를 rear 뒤에 연결, rear 갱신
- LQUEUE_DEQUEUE(q):
  front 노드 삭제, front 갱신, 기존 front 데이터 반환

노드

- 데이터와 다음 노드 정보의 묶음

  val: 저장된 데이터

  next: 다음 노드 주소 (마지막 노드는 NULL)

- NODE_CREATE(value, next_ptr):
  새 노드를 만들고 정보 설정

LQUEUE_ENQUEUE(q, value)

- 데이터가 value인 노드를 만들어 rear 뒤에 추가한다
- 동작: node에 NODE_CREATE(value, NULL)를 저장

  만약: q의 rear가 NULL이면

    q의 front와 q의 rear에 node를 저장한다

  그외:

    q의 rear의 next에 node를 저장한다

    q의 rear에 node를 저장한다
- 시간 효율: 상수 시간

LQUEUE_DEQUEUE(q)

- q의 첫 노드를 삭제하고 데이터를 반환한다
- 조건: q의 front 존재
- 동작: tmp에 q의 front를 저장한다
  q의 front에 tmp의 next를 저장한다
  만약: q의 front가 NULL이면 // 큐가 비었으면
    q의 rear에 NULL을 저장한다
  value에 tmp의 val을 저장한다
  tmp를 삭제한다
  반환: value
- 시간 효율: 상수 시간

배열로 구현한 큐

- 배열 인덱스의 범위는 1..capacity

  data: 데이터를 저장할 배열
  (크기는 capacity + 1, 인덱스 0 사용 안 함)
  capacity: 배열의 최대 저장량

  front: 가장 오래된 데이터의 인덱스 (초기값 1)
  rear: 가장 최신 데이터의 인덱스 (초기값 0)

배열로 구현한 큐

- SIMPLE_AQUEUE_ENQUEUE(q, value):

  rear 뒤에 데이터 value를 추가

- SIMPLE_AQUEUE_DEQUEUE(q):

  front 데이터 삭제 및 반환

SIMPLE_AQUEUE_ENQUEUE(q, value)

- q의 끝에 데이터 value를 추가한다
- 조건: q의 rear < q의 capacity
- 동작: q의 rear를 1 증가시키고 그 자리에 value를 저장한다
- 시간 효율: 상수 시간

SIMPLE_AQUEUE_DEQUEUE(q)

- q의 첫번째 데이터를 삭제하고 데이터를 반환한다
- 조건: q의 front ≤ q의 rear
- 동작: q의 front의 데이터를 tmp에 저장한다
  만약: q의 front와 q의 rear가 같으면
    q의 front에 1을, q의 rear에 0을 저장한다
  그외:
    q의 front를 1 증가시킨다
- 반환: tmp
- 시간 효율: 상수 시간

원형 배열로 구현한 큐

- 원형 배열: 인덱스가 capacity를 넘으면 다시 1이 됨
- 비어있음: front == rear + 1

  또는 (front == 1 && rear == capacity)
- 꽉차있음: front == (rear + 2)

  또는 (front == 1 && rear == capacity - 1)

  또는 (front == 2 && rear == capacity)
- 중간에 한 칸을 항상 띄워두어 비어있음과 꽉차있음을 구분한다

CIRCULAR_AQUEUE_EMPTY(q)

- q에 저장된 데이터가 없는지 확인한다

- 동작: 만약: q의 front와 q의 rear + 1이 같으면

  반환: 1

  만약: q의 front는 1이고,

  q의 rear가 q의 capacity와 같으면

  반환: 1

  반환: 0

CIRCULAR_AQUEUE_SIZE(q)

- q에 저장된 데이터의 개수를 반환한다
- 동작: 만약: q가 비어있으면

  반환: 0

  size에 rear - front + 1저장

  만약: size < 0

  size = size + capacity

  반환: size
- 특징: capacity - 1개를 꽉 찬 것으로 간주한다.

CIRCULAR_AQUEUE_ENQUEUE(q, value)

- q의 rear 뒤에 데이터 value를 추가한다

- 조건: q가 꽉 차있지 않다
- 동작: q의 rear를 1 증가시킨다

  만약: q의 rear > q의 capacity

  q의 rear에 1을 저장한다

  q의 data[q의 rear]에 value를 저장한다

- 시간 효율: 상수 시간

CIRCULAR_AQUEUE_DEQUEUE(q)

- q의 front 데이터를 삭제하고 반환한다
- 조건: q가 비어있지 않다
- 동작: tmp에 q의 data[q의 front]를 저장한다

  q의 front를 1 증가시킨다

  만약: q의 front > q의 capacity

  q의 front에 1을 저장한다

  tmp를 반환한다
- 시간 효율: 상수 시간

예제: 미로 찾기 문제

- 방문할 곳을 큐에 저장하면, 최단거리가 계산된다
- 경로 복원을 위해 이전 좌표 정보를 저장한다

미로 예시:

```
**********
*s       **
** * ** **
*    *    *
**** *** *
*        d*
**********
```

미로 정보: 2차원 배열 (배열의 배열)

- 미로[i][j]: (행 i, 열 j) 위치의 상태
- 0 : 통로(아직 처리하지 않음)
- 1 : 벽
- row*len + col : 최단경로에서 이전 좌표의 선형 index
- 0행과 0열은 모두 벽(1)로 두어, 내부 좌표에서 0/1과의 값 충돌을 방지함

좌표 구조체

- row: 행 좌표
- col: 열 좌표
- distance: 출발점부터의 거리 (출발점은 0)

STATUS(maze, point)

- 반환: maze[point의 row][point의 col]

- // 0 : 미방문 통로
- // 1 : 벽
- // ≥ len+1 : 최단거리 계산됨 - 이전 좌표의 선형 index
- // -1 : 출발점 표시

MARK(maze, point, from)

- maze[point의 row][point의 col]에 from 저장

- 최단경로에서 이전 좌표가 (i, j)일 때 from == i * len + j

  // 일차적으로는 처리가 됐다는 정보가 저장됨

좌표 선형 큐

- points: 좌표 구조체의 배열
- rear: 마지막으로 저장된 좌표의 인덱스
- front: 꺼낼 좌표의 인덱스

CREATE(cap)

- q에 새 좌표 선형 큐 저장
- q의 points에 새 배열 저장 (크기: cap + 1)
- q의 rear에 0 저장 (저장된 데이터 없음)
- q의 front에 1 저장 (처음 꺼낼 좌표)
- 반환: q

ENQUEUE(q, point)

- q의 points[q의 rear + 1]에 point 저장
- q의 rear 증가

DEQUEUE(q)

- 조건: q의 front ≤ q의 rear // 비어있지 않음
- tmp에 q의 points[q의 front] 저장
- q의 front 1 증가
- 반환: tmp

미로찾기 함수

- PUSH_NBRS(maze, len, to_visit, point)

미로 maze에서 point의 이웃 점들 중에 아직 확인하지 않은 점들을 큐 to_visit에 추가한다.

- FIND(maze, len, start, end)

미로 maze에 출발점 start에서의 최단 경로 정보를 저장한다.

- DECODE_PATH(maze, len, end, path_len)

도착점 end에 도달하는 최단 경로를 반환한다

PUSH_NBRS(maze, len, to_visit, point)

- 현재 위치 point의 인접 좌표 중에서 아직 갈 계획도 없었던 좌표들을 to_visit에 추가한다
- maze: 미로 정보 (갈 수 없거나 확인한 점이 표시됨)
- to_visit: 갈 곳들(큐)
- point: 현재 위치
- 이동은 상/하/좌/우 4-이웃만 허용

PUSH_NBRS(maze, len, to_visit, point)

- point_index에 point의 row * len + point의 col 저장
- 반복: point의 상하좌우 좌표 next에 대하여

  만약: STATUS(maze, next) == 0 이면

  next의 distance에 point의 distance + 1 저장

  MARK(maze, next, point_index)

  ENQUEUE(to_visit, next)

STATUS/MARK 시점 정리

- STATUS==0인 이웃만 후보로 본다
- 후보를 큐에 넣기 전에 MARK=point_index로 바꿔 '방문 예정' 표시
- 효과: 중복 ENQUEUE 방지, 최단 경로 정보 저장

FIND(maze, len, start, end)

- start에서 end까지 경로 정보를 maze에 저장한다, 경로의 거리를 반환한다
- maze: 미로 정보 (갈 수 없거나 확인한 점이 표시됨)
- start: 출발점 좌표
- end: 도착점 좌표
- len: 정사각형 미로의 한 변 길이 (행과 열 개수)

FIND(maze, len, start, end) 변수

- to_visit: 갈 곳 후보 큐(출발점에서 최단거리 이웃부터 처리)
- maze에 표시된 좌표는 to_visit에 추가하지 않음
- maze에 표시된 좌표보다 더 가까운 경로 없음

FIND(maze, len, start, end) 1/2

- to_visit에 CREATE(len * len) 저장 // 갈 곳들
- start의 distance에 0 저장
- MARK(maze, start, -1)
- ENQUEUE(to_visit, start)
- 반복: to_visit이 비어있지 않음

FIND(maze, len, start, end) 2/2

point에 DEQUEUE(to_visit) 저장

만약: point와 end의 좌표가 동일

반환: point.distance

PUSH_NBRS(maze, len, to_visit, point)

- 반환: -1 (못찾음)

maze에서 최단 경로 복원하기

- 경로를 저장할 배열을 충분한 크기로 만든다
- 도착점부터 거꾸로 가며 배열에 좌표를 저장한다

DECODE_PATH(maze, len, end, path_len)

- path에 새 좌표 배열 저장 (크기: path_len + 1)
- point에 end 저장
- 반복: i를 path_len부터 1까지

  data에 STATUS(maze, point)를 저장한다

  point와 path[i]의 row에 data / len를 저장한다

  point와 path[i]의 col에 data % len를 저장한다
- 반환: path

미로 찾기 복잡도 분석

- 시간: len^2 에 비례. 각 칸은 최대 한 번 방문/마킹
- 공간: len^2 에 비례. 큐(to_visit) + 경로 배열

요약

- 큐: 오래된 데이터부터 꺼내는 자료구조
- 단방향 링크드 리스트/배열로 구현 가능
- 모든 기능을 상수 시간으로 구현 가능
- 미로 찾기: 큐를 사용해서 최단 경로 탐색

단일스레드와 멀티스레드

- 단일스레드: 계산 장치가 있고, 계산 장치에 딸린 고속/소형 메모리가 있고, 일반 메모리가 있다.
- 계산을 할 때는 일반 메모리에서 필요한 부분을 고속/소형 메모리에 복사해 놓고 작업한다.
- 다른 계산을 할 때는 고속/소형 메모리에서 다시 일반 메모리에 계산 결과를 복사한다.

단일스레드와 멀티스레드

- 멀티스레드: 계산 장치가 여러개 있고, 계산 장치 각각에 딸린 고속/소형 메모리가 있고, 일반 메모리는 공유한다.
- 계산을 할 때는 일반 메모리에서 필요한 부분을 각자의 고속/소형 메모리에 복사해 놓고 작업한다.
- 나중에 각자의 고속/소형 메모리에서 다시 일반 메모리에 계산 결과를 복사한다.

멀티스레드 문제: 동시에 값을 읽으면?

- 초기값이 0인 전역 변수 x의 값을 두 스레드에서 1 증가시키고 있는데..
- 운 좋은 케이스 - 한 스레드가 x의 값을 1로 바꾸고, 다른 스레드는 x의 값을 2로 바꾼다
- 다른 케이스 - 한 스레드가 x의 값을 읽어서 1을 더하는 중에 다른 스레드도 x의 값을 읽었다면, 두 스레드 모두에서 계산 결과는 1이고, 메모리에 1만 두번 저장됨

멀티스레드에서의 원자성

- 초기값이 0인 전역 변수 x의 값을 두 스레드에서 1 증가시키고 있는데..
- x에 원자성을 부여하면, 원자성이 적용된 연산을 쓸 수 있다. 결과는 항상 2가 된다
- 원자: 더이상 쪼개지지 않는 단위. 1 증가시키는 원자성 함수 - 중간에 다른 스레드가 값에 접근할 수 없다

- 한 스레드가 ++을 하기 위해 메모리에서 x의 값을 캐시에 복사해놓고, 1을 더해서 캐시에 저장하고, 다시 메모리에 저장할 때까지 다른 스레드에서 메모리의 x에 접근할 수 없다.

원자성 예제 1/2

- 원자성 전역 변수 acnt
- 일반 전역 변수 cnt

- 함수 f:

  반복: 1000회

  원자성_누적_덧셈(acnt, 1)

  ++cnt // data race로 인한 누락 발생

원자성 예제 2/2

- 스레드 배열 thr (크기: 11)

- 반복: n이 1부터 10까지
  스레드_만들기(thr[n], f)

- 반복: n이 1부터 10까지
  스레드_종료_기다리기(thr[n])

- 출력(acnt) // 10000이 출력됨
- 출력(cnt) // 9511이 출력됨

원자성 배열 큐

- 전역 변수 q[N]
- 원자성 전역 변수 rear
- 함수 ENQUEUE(x):

  my_slot = 원자성_누적_덧셈(rear, 1)

  q[my_slot] = x

원자성 함수

- 원자성_누적_덧셈(obj, arg):

  obj = obj + arg를 원자성으로 실행

- z = 원자성_교체(obj, desired):

  z = obj, obj = desired를 원자성으로 실행

원자성 함수

- 원자성_조건부_교체(obj, expected, desired):

  만약: obj와 expected를 원자성 비교해서 같으면

    obj = desired를 원자성으로 실행

  그외:

    expected = obj를 원자성으로 실행

원자성 링크드 리스트

- node 구조체: value, next
- 원자성 전역 변수 first
- 함수 INSERT_FIRST(x):

  new_n = 새로운 node

  new_n의 value = 새로운 node

  old_first = 원자성_읽기(first)

원자성 링크드 리스트

반복실행:

new_n의 next = old_first

반복조건: 원자성_조건부_교체(

first, old_first, new_n)이 거짓

원자성 링크드 리스트 큐

- pointer_t 구조체: ptr /*node_t 포인터*/, count
- node_t 구조체: value, next /*pointer_t*/
- queue_t 구조체: Head /*pointer_t*/, Tail /*pointer_t*/
- 함수 INITIALIZE (Q /*queue_t 포인터*/):

  node = 새로운 node_t

  node->next.ptr = NULL

  Q->Head = Q->Tail = node

원자성 링크드 리스트 큐

함수 ENQUEUE(Q /*queue_t 포인터*/, value):

- node = 새로운 node_t
- node->value = value
- node->next.ptr = NULL
- 반복:

  tail = Q->Tail

  next = tail.ptr->next

원자성 링크드 리스트 큐

만약: tail == Q->Tail

만약: next.ptr == NULL

만약: CAS (&tail.ptr->next, next,

<node, next.count+1>)

반복 종료

원자성 링크드 리스트 큐

그외: // next.ptr != NULL일 때

CAS(&Q->Tail, tail,

<next.ptr, tail.count+1>)

- CAS(&Q->Tail, tail, <node.ptr, tail.count+1>)

원자성 링크드 리스트 큐

함수 DEQUEUE (Q /*queue_t 포인터*/,
pvalue /*데이터 타입 포인터*/):

- 반복:

  head = Q->Head

  tail = Q->Tail

  next = head->next

원자성 링크드 리스트 큐

만약: head == Q->Head

만약: head.ptr == tail.ptr

만약: next.ptr == NULL

반환: 거짓

CAS(&Q->Tail, tail,

<next.ptr, tail.count+1>)

원자성 링크드 리스트 큐

그외: // head.ptr != tail.ptr

*pvalue = next.ptr->value

만약: CAS(&Q->Head, head,

<next.ptr, head.count+1>)

반복 종료

- free(head.ptr)
- 반환: 참

출처

- 멀티스레드: 유튜브 CppCon 2017: Fedor Pikus “C++ atomics, from basic to advanced. What do they really do?”
- 원자성 예제 코드: https://cppreference.com/w/c/atomic/atomic_fetch_add.html
- 멀티스레드 링크드 리스트 큐: https://cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf