data_tructures(&algorithms, lecture07)

Doan Trung Tung, PhD – University of Greenwich (Vietnam)

Plan

Queue ADT

Introduction, queue operation, queue a
pplication

Array Implementation

How to implement queue with Array, Circular
Queue with array

Linked List Implementation

How to implement queue with Linked Li
st

Priority Queue

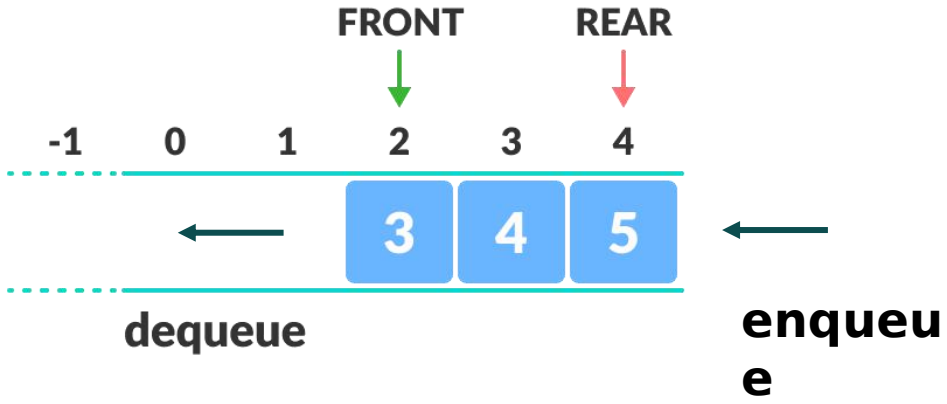
What is Priority Queue, implement by Ar
ray and by Heap

Queue ADT

Introduce Queue data structure, some operations

What Is A Queue

- A linear data structure that can be accessed in both begins and ends for storing and retrieving data
- A queue is a First In, First Out (FIFO) data structure



Operations On Queue

- clear: clear the queue
- is empty: check if a queue is empty
- enqueue: put element on the rear of the queue
- dequeue: take the element on the top out of the queue
- front: get first element without removing it
- rear: get last element without removing it
- size: get size of the queue

Applications Of Queue

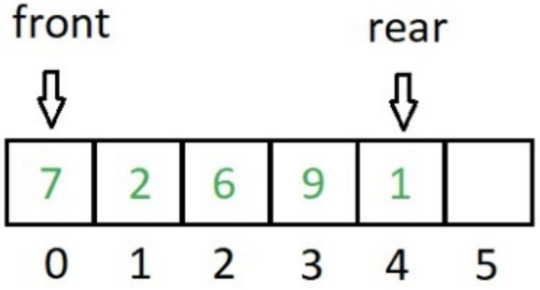
- Any kind of waiting list
- Operating System:
  - CPU scheduling
  - Disk scheduling
  - Shared resources between processes
- Traverse in tree / graph data structures
- Turned-base games
- Alt-Tab in Windows
- ...

Array Implementation

Implement Queue by Array, Circular Queue

Queue Interface By Array

- Need global variables
- 2 indicators run forward
- Queue size is fixed
- Number of



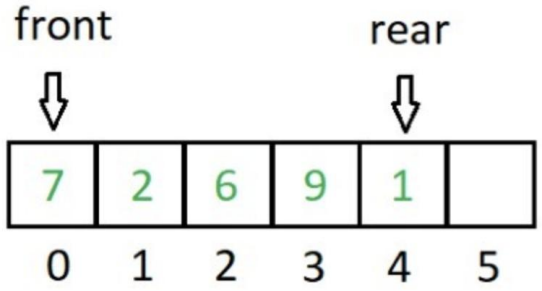

```
#define QUEUE_SIZE 100

extern int front;
extern int rear;
extern int queue[];

void clear_queue(void);
int is_empty(void);
int is_full(void);
void enqueue(const int n);
int dequeue(void);
int size(void);
void overflow(char* msg);
```

Queue Implementation By Array

- is_empty: when front = -1
- is_full: when rear runs to the end of array
- enqueue: need to check full queue before, then increase rear and assign new item to new rear position
- dequeue: need to check empty queue before, then increase front and return the item at old front
- size: distanc ar

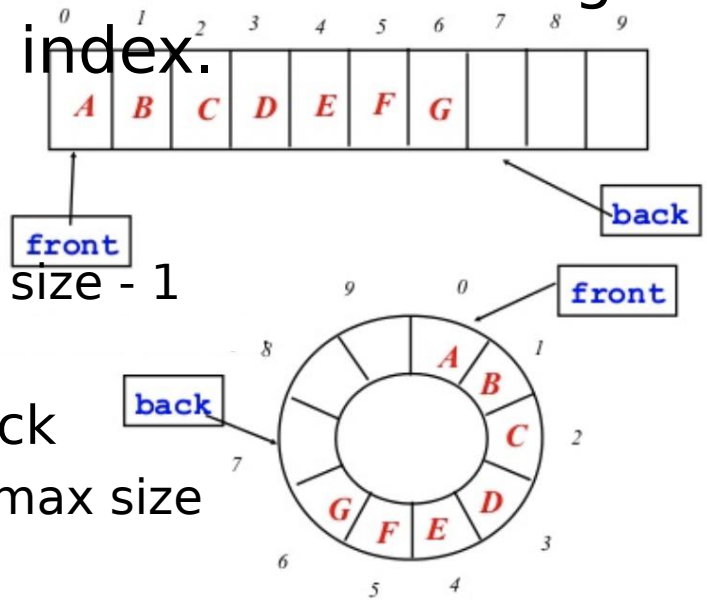


Queue Implementation By Array

- Number of enqueue is fixed
  - => When dequeue, if queue has only one item, clear queue to reset front & rear to -1
  - => Implement circular queue
- Size is fixed
  - => Using dynamic array and grow size if queue is full

Circular Queue By Array

- Main idea: front & rear run forward but go back (if possible) to 0 index.
  - is_empty: front = -1
  - is_full:
    - front = 0, rear = max size - 1
    - front = rear + 1
  - Using modulo to go back
    - front = (front + 1) % max size
  - real size:
    - if (front < rear): rear – front + 1
    - else: max size - (front - rear - 1)



Example: Queue of messages

- Simulation of sending message. Commands:
  - +message: enqueue message
    - When the queue is full, all messages will be sent
  - -: dequeue a message to send
  - a: Sending all messages by dequeue one by one
  - q: Quit simulation, all messages will be sent
- Using Circular Queue will support unlimited enqueue operation, still having full condition

Example: Queue of messages

- Queue: array of char pointers in heap memory
- Enqueue

```
Check if queue is full
Else
    If queue is empty front = 0
    Increase rear (go back if needed)
    Add message to rear position
```

- Dequeue

```
Check if message is empty
Else
    Get message at front
    If there is one message, clear queue
    Else increase front (go back if needed)
```

Example: Queue of messages

- Running simulation

```
Message sending simulation
>> +hello
Saved hello
>> +world
Saved world
>> +stack and queue
Saved stack and queue
>> -
Sending hello ...
>> +good bye
Saved good bye
>> a
Sending world ...
Sending stack and queue ...
Sending good bye ...
>> q
No message to send!
Program ended with exit code: 0
```

Linked List Implementation

Implement Queue by Linked List

LinkedList vs Queue

- Queue is Linked List with limited operations and/or other interfaces

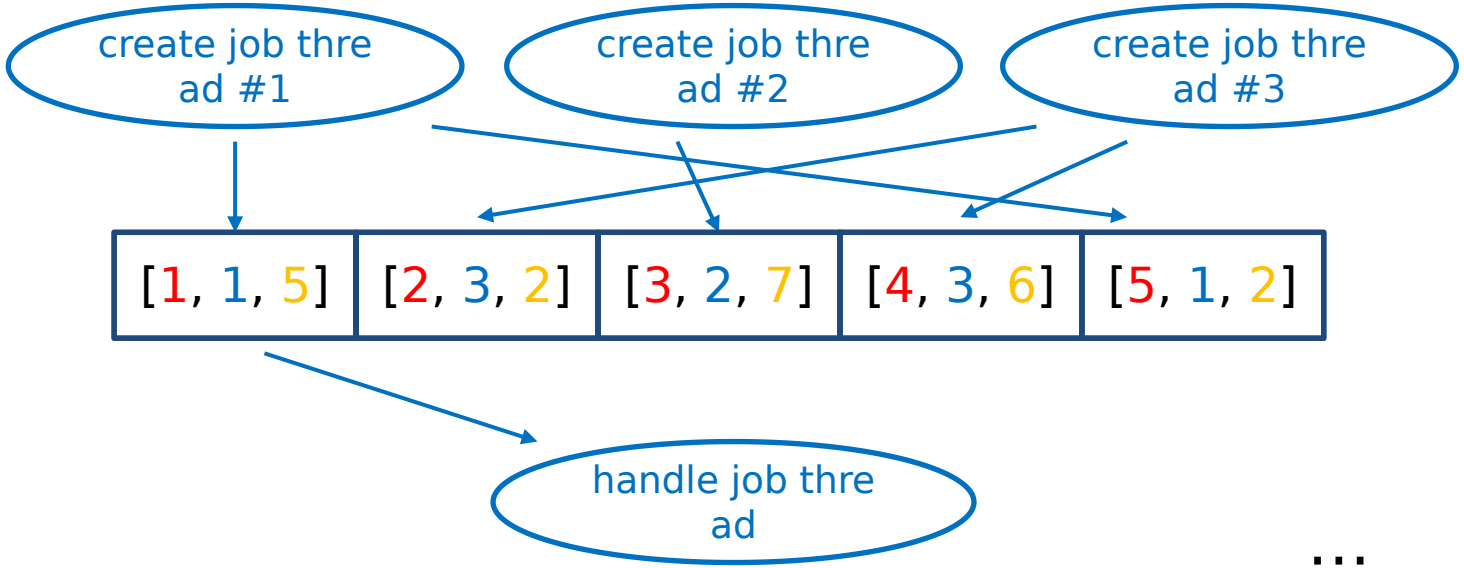
Linked List	Queue
Add to end	Enqueue
Remove from head	Dequeue
Clear list	Clear queue
Is empty	Is empty
Get size	Get size
	To array
	Front / Rear

Example: Queue of Jobs

- Simulation a queue of jobs
  - A job has an id, agent id (who creates this job) and duration (in seconds)
  - Several threads will create jobs in parallel and store them in a job queue
  - A thread will get job from that queue to execute it
  - Shared resources: queue of jobs and number of jobs

Example: Queue of Jobs

- Simulation a queue of jobs



Example: Queue of Jobs

- Implement queue of jobs
  - Job data

```
typedef struct
{
    int job_id;
    int agent_id;
    int duration; // in second
} job_data;
```

Job node in LL

```
typedef struct str_job job;

struct str_job
{
    job_data data;
    job* next;
};
```

Example: Queue of Jobs

- Implement queue of jobs
  - Enqueue operation

```
void enqueue(job** jobs_queue, job_data ajob)
{
    job* j = create_job(ajob);

    if (is_empty(*jobs_queue)) *jobs_queue = j;
    else
    {
        job* rear = *jobs_queue;
        while (rear->next != NULL) rear = rear->next;
        rear->next = j;
    }
}
```

Example: Queue of Jobs

- Implement queue of jobs
  - Dequeue operation

```
job_data dequeue(job** jobs_queue)
{
    if (is_empty(*jobs_queue))
    {
        printf("Jobs queue is empty!");
        exit(1);
    }
    else
    {
        job* j = *jobs_queue;
        *jobs_queue = j->next;
        job_data data = j->data;
        free(j);
        return data;
    }
}
```

Example: Queue of Jobs

- Implement parallel threads
  - pthread_create: create a new thread to run in parallel

```
pthread_create(&tid, NULL, handle_jobs, NULL);
```

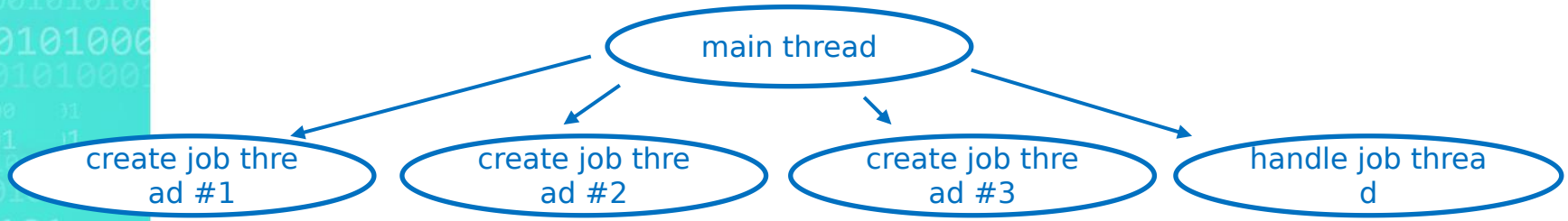
thread structure

thread attribute

callback function

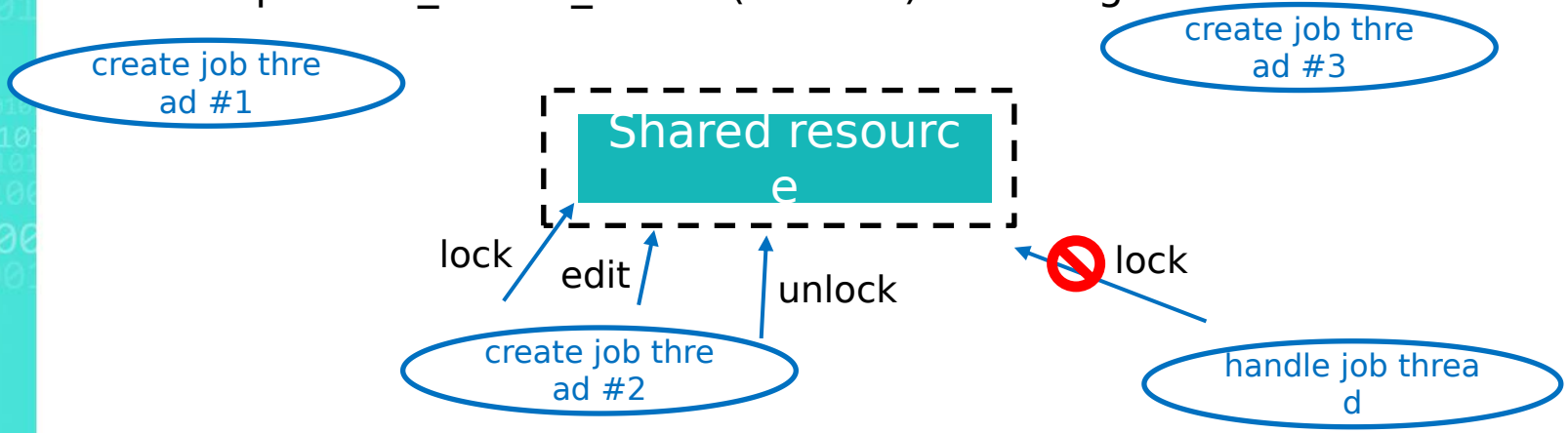
arguments

  - pthread_exit: terminate a thread
  - sleep: to make current thread idle for a while



Example: Queue of Jobs

- Shared resources
  - pthread_mutex_t locker: declare a locker to lock shared resource
  - pthread_mutex_init(&locker,NULL): initialize the locker
  - pthread_mutex_lock(&locker): lock global resources
  - pthread_mutex_unlock(&locker): unlock global resources



Example: Queue of Jobs

- Putting all together
  - main thread

```
// init locker
pthread_mutex_init(&locker,NULL);

// create threads to generate jobs in parallel
for (int agent_id = 1; agent_id <= NAGENTS; agent_id++)
    pthread_create(&tid, NULL, generate_jobs, (void*)agent_id);
// create thread to handle generated jobs
pthread_create(&tid, NULL, handle_jobs, NULL);
```

Example: Queue of Jobs

- Putting all together
  - create job thread

```
int agent_id = (int) threadid;
for (int i = 1; i <= NJOBS; i++)
{
    pthread_mutex_lock(&locker); // lock the locker

    // increase number of jobs
    // create job data based on agent id, number of jobs
    // enqueue job data

    pthread_mutex_unlock(&locker); // unlock the locker

    sleep(rand() % WAIT_TIME);
}
pthread_exit(NULL);
```

Example: Queue of Jobs

- Putting all together
  - handle job thread

```
Endless loop
  Lock the locker
  If jobs queue is empty and no more jobs in future
        Quit handle thread
  Else If queue is not empty
        Handle a job
  Else
        Wait for a job
```

Example: Queue of Jobs

- Putting all together

```
Agent id 1 create job 1
Number of waiting jobs: 1
Handle job 1 of agent 1 in 4s
Agent id 2 create job 2
Agent id 3 create job 3
Agent id 2 create job 4
Agent id 2 create job 5
Agent id 1 create job 6
Agent id 3 create job 7
Number of waiting jobs: 6
Handle job 2 of agent 2 in 0s
Number of waiting jobs: 5
Handle job 3 of agent 3 in 0s
Number of waiting jobs: 4
Handle job 4 of agent 2 in 4s
```

Priority Queue

Introduce Priority Queue and how to implement it

What is Priority Queue?

- Same as a normal queue but …
  - Each element has a priority
  - Elements with higher priority will be dequeued before elements with lower priority
  - Elements with same priority will be dequeued in the order of enqueuing

> “**All animals are equal, but some animals a re more equal than others**”
>
> *George Orwell – Animal Farm*

Example of Priority Queue

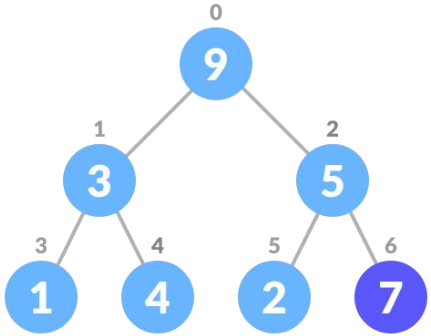
- Enqueue following pairs (key, priority)
  - (1, 1), (2, 1), (3, 2), (4, 1), (5, 2), (6, 3), (7, 1)
- Actual queue in building:
  - (1, 1)
  - (1, 1), (2, 1)
  - (3, 2), (1, 1), (2, 1)
  - (3, 2), (1, 1), (2, 1), (4, 1)
  - (3, 2), (5, 2), (1, 1), (2, 1), (4, 1)
  - (6, 3), (3, 2), (5, 2), (1, 1), (2, 1), (4, 1)
  - (6, 3), (3, 2), (5, 2), (1, 1), (2, 1), (4, 1), (7, 1)

Priority Queue by Array

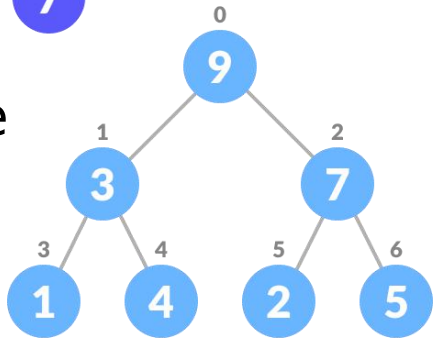
- Enqueue algorithm:
  - Insert new element at the rear
  - If priority of new element is greater than the one ahead then move it up until it’s less than or equal
    - (3, 2), (1, 1), (2, 1), (4, 1)
    - (3, 2), (1, 1), (2, 1), (4, 1), (5, 2)
    - (3, 2), (1, 1), (2, 1), (5, 2), (4, 1)
    - (3, 2), (1, 1), (5, 2), (2, 1), (4, 1)
    - (3, 2), (5, 2), (1, 1), (2, 1), (4, 1)
  - Complexity:
    - Enqueue: O(n)
    - Dequeue: O(1)

Priority Queue by Heap

- Enqueue algorithm:
  - Insert new element at the end of the heap

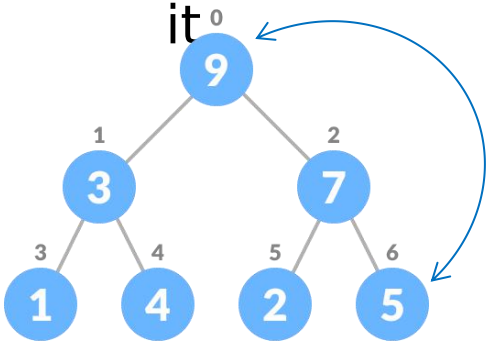


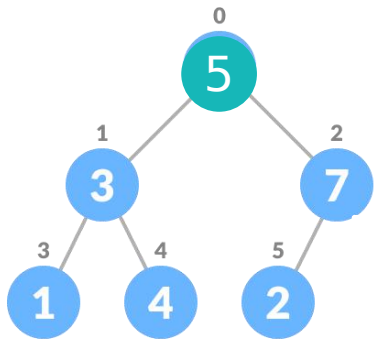
  - Heapify the whole tree

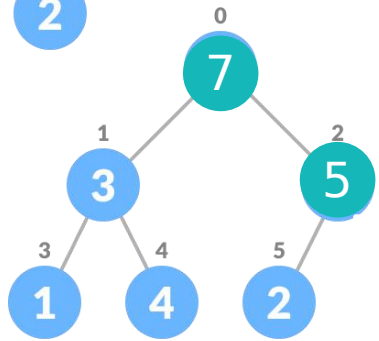


Priority Queue by Heap

- Dequeue algorithm:
  - Swap the root with the last element then remove it







  - Heapify the whole tree
  - Complexity:
    - Enqueue: O(logn)

Priority Queue by Heap

- Implement Priority Queue by Heap

```
typedef struct
{
    int key;
    int priority;
    int order;
} element;
```

```
extern int qsize;

void swap(element *a, element *b);
void print_queue(element* queue);
void heapify(element* queue, int i);
void enqueue(element* queue, element e);
element dequeue(element* queue);
int greater(element a, element b);
int is_empty(void);
```

Fix unstable characteristics of heap

Priority Queue by Heap

- Implement PriorityQueue by Heap

```
void heapify(element* queue, int i)
{
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int imax = i;

    if (left < qsize && greater(queue[left], queue[imax]))
        imax = left;
    if (right < qsize && greater(queue[right], queue[imax]))
        imax = right;

    if (imax != i)
    {
        swap(&queue[i], &queue[imax]);
        heapify(queue, imax);
    }
}
```

Priority Queue by Heap

- Implement Priority Queue by Heap

```
int greater(element a, element b)
{
    // First compare by priority
    // Then compare by order if same priority
}
```

```
void enqueue(element* queue, element e)
{
    // Add e to the end of the queue
    // Heapify the whole tree bottom-up
}
```

```
element dequeue(element* queue)
{
    // Get first element, save into e
    // Swap first element and last element
    // Heapify the new tree (exclude last element)
    // Return e
}
```