CS 261

Dynamic Array Queue
by
Tim Budd
Ron Metoyer
Sinisa Todorovic
Dynamic Array -- Review

• Positives:
  – Each element easily accessed
  – Grows as needed
  – The user unaware of memory management
Stack as Dynamic Array -- Review

• Remove and add elements from top
• Occasional capacity increase
• Remove operation has complexity $O(1)$
• Add operation has complexity $O(1)+$
  – $O(1)$ expected
  – $O(n)$ worst case
Bag as Dynamic Array -- Review

- Order is unimportant, so adding to end
- Remove *always* requires moving elements
- Add is $O(1)^+$
  - $O(n)$ worst case
- Remove is $O(n)$
Queue
Queue

- Elements are inserted at one end, and removed from another
- E.g. queue of people
- First in, first out (FIFO)
Interface View of Queue

- `addBack(newElement)` -- inserts an element
- `front()` -- returns the first element
- `removeFront()` -- removes the first element
- `isEmpty()` -- checks if the queue is empty
Queue as Dynamic Array

• Which end is better for insertion?
• Which end is better for removal?
• What would be \( O(\_\_\_\_) \) ?
Removing from Front, Adding to Back

Remove requires moving elements => $O(n)$

Insertion to the end is $O(1)+$
Removing from Back, Adding to Front

Insertion:

Insertion requires moving elements  \( \Rightarrow O(n) \)
Removal from the end is \( O(1) \)
Double-Ended Queue
= Deque
Deque

- Allows:
  - Insertions at both front and back
  - Removals at both front and back
- Also called, double-ended stack
Interface View of Deque

- `addFront(newElem)` -- inserts to the front
- `addBack(newElem)` -- inserts to the back
- `front()` -- returns the first front element
- `back()` -- returns the first back element
- `removeFront()` -- removes from the front
- `removeBack()` -- removes from the back
- `isEmpty()` -- checks if the queue is empty
Dequeue as Dynamic Array

• **Key idea:**
  – Do not tie "front" to index zero

• Instead,
  – allow both "front" and "back" to float around the array
Example

```
DataSize = 6
DataStart = 2
Data = [2, 4, 7, 3, 9, 1]
```

start  back_idx
Example

DataSize = 6
DataStart = 7
Data = 2 4 7 3

back_idx

start
Deque Implementation

```c
struct deque {
    TYPE * data;
    int capacity;
    int size;
    int start;
};
```
Keeping size vs Keeping pointer to end

- We compute the index of end from the index of front and size
- Why not keep the index of end?
- OK, but need to compute size frequently
Adding/Removing for Deque

• Add to front: back off starting point by 1

• Add to back: increase size by 1

• Remove from front: increase starting point by 1

• Remove from back: decrease size by 1
Adding/Removing for Deque

What if elements wrap around?

DataSize = 6
DataStart = 7
Data = 9 1 2 4 7 3
Wrapping: How to Compute New Index

- If Index < 0, then add capacity
- If Index > capacity, then subtract capacity
- If size == capacity, reallocate new buffer
Wrapping: How to Compute New Index

Use the **mod** operator:

\[
\text{backIndex} = (\text{da->start} + \text{da->size}) \mod \text{da->cap};
\]
Implementation
Deque Structure

```c
struct deque {
    TYPE * data;
    int capacity;
    int size;
    int start;
};
```
void initDeque (struct deque *d, int initCapacity) {
    d->size = d->start = 0;
    assert(initCapacity > 0);
    d->capacity = initCapacity;
    d->data =
        (TYPE *) malloc(initCapacity * sizeof(TYPE));
    assert(d->data != 0);
}
void _doubleCapDeque (struct deque *d) {
    TYPE * oldData = d->data;
    int oldStart = d->start;
    int oldSize = d->size;
    int oldCapacity = d->capacity;
    int j;
    initDeque(d, 2 * oldCapacity);
    for (j = 0 ; j < oldSize; j++) {
        d->data[j] = oldData[oldStart++];
        if (oldStart >= oldCapacity) oldStart = 0;
    }
    free(oldData);
    d->size = oldSize;
}
addBackDeque

```c
void addBackDeque(struct deque *d, TYPE val) {

    int back_idx;

    if (d->size >= d->capacity) _doubleCapDeque(d);

    /* Increment the back index */

    back_idx = (d->start + d->size) % d->capacity;

    d->data[back_idx] = val;

    d->size ++;
}
```
addFrontDeque

```c
void addFrontDeque(struct deque *d, TYPE val) {
    if (d->size >= d->capacity) _doubleCapDeque(d);

    /* Decrement the front index */
    d->start--; 
    if (d->start < 0) d->start += d->capacity;
    d->data[d->start] = val;
    d->size ++;
}
```

DataSize = 6
DataStart = 7
Data = 2 4 7 3

start
Worksheet 20

• Implement Dynamic Array Deque

• How do you
  – Add to front or back?
  – Return front? Return back?
  – Remove front? Remove back?
Dynamic Array -- Problems

• Data kept in a single large block of memory

• Often more memory used than necessary
  – especially when repeatedly growing and shrinking the dynamic array