CS 261: Data Structures

Dynamic Array Queue
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/to top
- Occasional capacity increase
- Remove operation has complexity $O(1)$
- Add operation has complexity $O(1)$
Bag as Dynamic Array -- Review

• Order is not important, so adding to the end
• Add is $O(1)$, with occasional capacity increase
• Remove is $O(n)$
Double-Ended Stack

= Deque
Deque

• Allows:
  – Insertions at both front and back
  – Removals at both front and back
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
Deque as Dynamic Array

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

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

In this example, start index is **larger** than back index

Data Size = 6
Data Start = 7
Data = 9 1 2 4 7 3

backIndex  startIndex
Adding/Removing for Deque

- **Add front**: decrease the start index by 1
- **Add back**: increase size by 1
Adding/Removing for Deque

- **Remove front**: increase the start index by 1
- **Remove back**: decrease size by 1
Adding/Removing for Deque

What if elements wrap around?

DataSize = 6
DataStart = 7
Data =

9 1 2 3 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

DataSize = 6
DataStart = 7
Data = 9 1

9 1 2 4 7 3
Implementation
Deque Structure

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

- We compute the back index from the start index and size
- Why not keep the back index?
- OK, but need to compute size frequently
Wrapping: How to Compute Back Index

Use the **mod** operator:

\[
\text{backIndex} = (\text{start} + \text{size}) \mod \text{capacity};
\]
void initDeque (struct deque *d, int initCapacity) {

d->size = d->start = 0; /* initially, no data in Deque */

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; /*memorize old data*/
    int oldStart = d->start; /*memorize old start*/
    int oldSize = d->size; /*memorize old size*/
    int oldCapacity = d->capacity; /*memorize old cap.*/
    int j;
    initDeque(d, 2 * oldCapacity); /*new memory alloc.*/
    for (j = 0 ; j < oldSize; j++) {/*copy back old data*/
        d->data[j] = oldData[oldStart++];
        if (oldStart >= oldCapacity) oldStart = 0;
    }
    free(oldData);
    d->size = oldSize;
}
```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 ++;
}
```

```c
DataSize = 6
DataStart = 7
Data =
```

![Deque Visualization]

9 1 _ _ _ 2 4 7 3

back_idx  start
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 = 9 1 2 4 7 3
Worksheet 20

• Implement Dynamic Array Deque

• How do you
  – Add to front or back?
  – Return front? Return back?
  – Remove front? Remove back?
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 $O(\cdot)$?
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 => O(n)

Removal from the end is O(1)
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