UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Instituto de Biociências, Letras e Ciências Exatas - Câmpus de São José do Rio Preto

Trabalho 1 ED PPGCC

Aluno: Jogi Suda Neto

Fila de prioridades

Uma fila de prioridades é basicamente uma extensão da estrutura de dados do tipo fila, com algumas propriedades adicionais: cada elemento a entrar na fila será inserido com algum valor de prioridade associado, e a remoção de um elemento desta mesma fila deve ser sempre daquele de máxima prioridade.

Propriedades adicionais em versões mais avançadas da fila de prioridades são:

- Modificar a prioridade da algum elemento da fila.
- Obter o número de elementos atuais na fila.
- Testar se há elementos de mesma prioridade na fila.

Aplicações incluem: gerenciamento de processos no computador de acordo com alguma ordem de prioridade; algoritmos gulosos; encaminhamento de pacotes de rede em ordem de urgência.

As filas podem ser feitas utilizando listas (simples ou duplamente encadeada) e heaps (a serem explicados adiante). Para as listas: os elementos na fila pode ser inseridos de maneira ordenada ou não, sendo que para o primeiro caso a complexidade de remoção (e seleção) do maior elemento será O(1), porém para a inserção essa complexidade não será mais O(1), mas sim O(n) ou até maior dependendo do algoritmo de ordenação utilizado. Também será O(n) para a alteração da prioridade de algum elemento, e a construção, em O(nlogn).

Para as listas não ordenadas, temos inserção em O(1), porém remoção, busca, alteração, construção e seleção todos em O(n). As implementações das funções **inserir**, **removeMax**, e **alterarPrio** para listas ordenadas são

descritas abaixo:

```
typedef struct node {
    int data;

    int priority;

    struct node* next;

} Node;

// criar novo node
Node* newNode(int d, int p)
{
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = d;
    temp->priority = p;
    temp->next = NULL;

    return temp;
}

// Retorna valor de maior prio: número const. de operações, logo, O(1)
int select(Node** head)
{
    return (*head)->data;
}

// Remove o elemento de maior prio: número const de operações, logo, O(1)
void removeMax(Node** head)
{
    Node* temp = *head;
    (*head) = (*head)->next;
    free(temp);
}

void alterarPrio(Node** head, int data, int newPrio){

/*altera prio de algum elemento: a lista é percorrida no máximo 2 vezes, logo,
temos O(n) para alterarPrio.
*/
      Node* start = (*head);

      while(start != NULL && (start->next)->data != data){
            start = start->next
      }
      if(start == NULL)
            error "nó não encontrado."
```

```
      Node *tmp = start->next; // desalocar o nó daquela antiga pos.
      start->prox = tmp->next;
      tmp->priority = newPrio;

      start = (*head);

      while(start->next != NULL && (start->next)->priority > newPrio){
            start = start->next
      }


      if(start->next == NULL){ //caso especial, nó realocado pro final
            start->next = tmp;
            tmp->next = NULL;
      }

      tmp->next = start->next;
      start->next = tmp;

}

void inserir(Node** head, int d, int p) {

*/a complexidade de inserir está no while, que percorrerá a lista no máximo
uma vez inteira, logo temos O(n)
*/


    Node* start = (*head);

    // Create new Node
    Node* temp = newNode(d, p);

    // Caso especial: nó a ser inserido têm prio maior que head.
    if ((*head)->priority < p) {

        // Insere novo nó antes da cabeça
        temp->next = *head;
        (*head) = temp;
    }
    else {

        // Caminha na lista e acha a posição correta para inserir
        while (start->next != NULL &&
               start->next->priority > p) {
            start = start->next;
        }

        // ou está no fim da lista, ou na posição encontrada
        temp->next = start->next;
        start->next = temp;
    }
}
```

Para a implementação da versão não ordenada das listas, bastaria

inserir cada novo elemento sempre no final da lista, porém na remoção não teríamos acesso instantâneo, pois teríamos que buscar a maior prioridade na lista.

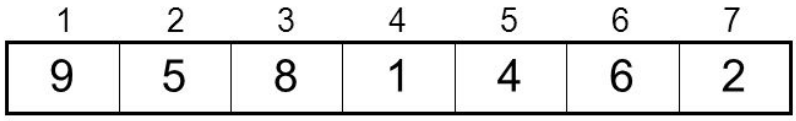
Abaixo será introduzido a estrutura de dados Heap, que apresentará melhoras em relação à implementação da fila de prioridades utilizando listas, sendo O(1) para seleção do elemento de máxima prioridade, O(logn) para inserção, remoção e alteração, e O(nlogn) para construção desta nova estrutura.

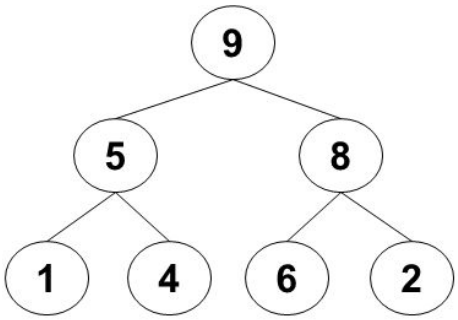
Heap binário

Heap binário é basicamente uma estrutura de dados subcaso de uma árvore binária completa ou quase completa (pelo menos o penúltimo nível deve estar completo) que obedece a seguinte relação de ordem: cada nó pai deve ser **maior ou igual** a seus filhos. A esta estrutura chamamos de max-heap, sendo que podemos ter também uma min-heap: basta considerar a relação de ordem em que cada pai seja **menor ou igual** a seus filhos. Em um vetor de n elementos, temos um max-heap se: $A[\lfloor (i/2) \rfloor] \geq A[i]$, para i = 1, 2, 3,... ,n.

Heap Binário Máximo

- *Heap* Binário tal que um nodo pai tem valor maior ou igual ao valor dos nodos filhos





Operações básicas do heap são:

- **criarHeap**: deve criar e retornar um heap vazio.
- **buscaMax**: recebe uma heap de entrada, e deve buscar e retornar seu maior elemento (assumindo que seja um max-heap, caso contrário

teríamos a função análoga buscaMin, que deve buscar e retornar o menor elemento da heap).

- **insertHeap**: dado uma heap de entrada, insere o novo elemento com sua chave de prioridade associada, e então rebalanceia a heap.
- **removeMax**: remove o maior elemento da heap.
- **altPrio**: altera a prioridade de um dado elemento, sendo que ao final, a heap deve ser rebalanceada.
- **mergeHeap**: dado duas heaps de entrada, combina as duas em uma só. A heap final deve ser rebalanceada.

Implementações

```
buscaMax(A){
        return A[1];
}
```

```
removeMax(A){
        se tamanhodoHeap[A] < 1
                erro "heap underflow"
        max = A[1]
        A[1] = A[tamanhodoHeap[A]]
        tamanhodoHeap[A] -= 1
        maxHeapify(A, 1)
}
```

```
altPrio(A, i, chave){
        se chave < A[i]
                erro "impossível inserir uma chave menor que a atual"
        A[i] = chave
        enquanto i > 1 e A[PAI(i)] < A[i]
                faça troca(A[i], A[PAI(i)])
                        i = PAI(i)
}
insertHeap(A, chave){
        tamanhoHeap[A]++
        A[tamanhoHeap[A]] = -inf        //infinito negativo
        incChave(A, tamanhoHeap[A], chave)
}
```