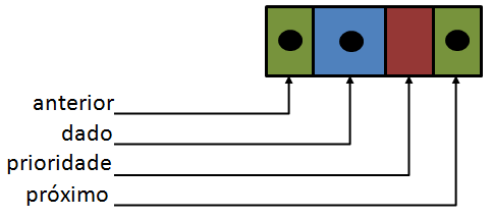
Fila de Prioridade

A fila de prioridade nada mais é que uma fila comum que permite que elementos sejam adicionados associados com uma prioridade. Cada elemento na fila deve possuir um dado adicional que representa sua prioridade de atendimento. Uma regra explicita define que o elemento de maior prioridade (o que tem o maior número associado) deve ser o primeiro a ser removido da fila, quando uma remoção é requerida.

As filas de prioridade são utilizadas em diversas situações. Por exemplo nas filas do banco existe um esquema de prioridade em que clientes preferenciais, idosos ou mulheres grávidas possuem uma mais alta prioridade de atendimento se comparados aos demais clientes. Em filas de atendimento para análise de sinais em sistemas de controle, sensores prioritários são adicionados a fila de atendimento com uma maior prioridade que sensores secundários, o que garante o atendimento de sinais prioritários a frente dos demais.

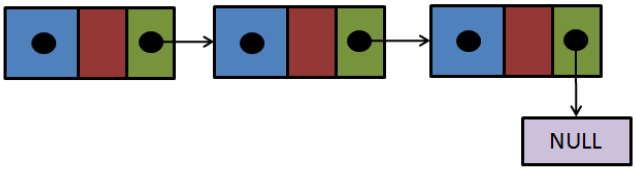
Estrutura

Como discutido anteriormente, os elementos ou células de uma fila de prioridade

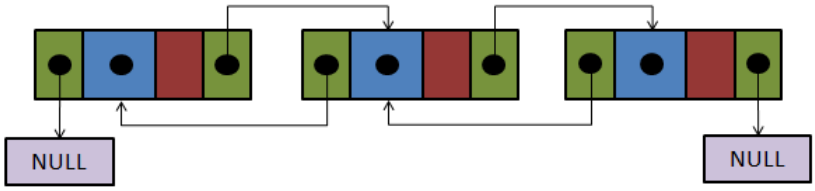



```
struct cel{
  struct cel *ant;
  struct cel *prox;
  int dado;
  int prioridade;
};
typedef struct cel celula;
```

As células em uma fila podem ser encadeadas de maneira simples ou dupla tal como os diagramas apresentados abaixo. A escolha do método de encadeamento deve ser dirigida pela necessidade imposta pela aplicação a qual a estrutura de dados se presta assim como por outros fatores tal como o fato dos dados serem inseridos de maneira ordenada ou não;



Fila de prioridade usando uma lista encadeada simples



Fila de prioridade usando uma lista duplamente encadeada

Operações

As filas de prioridade prevêem duas operações básicas que são na realidade uma extensão das operações básicas de uma fila comum. São elas:

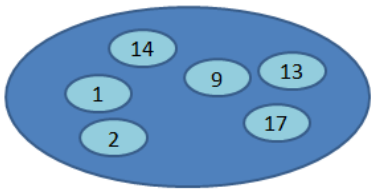
- Inserir com prioridade
- Remover elemento de mais alta prioridade

Adicionalmente, versões mais avançadas de filas de prioridade podem requerer as seguintes operações:

- Alterar prioridade de um dado elemento;
- Número de elementos na fila;
- Testar a existência de elementos de mesma prioridade;

Tópico correlato: Encontrar o maior elemento de uma lista

Antes de começarmos a estudar como implementar filas de prioridade é necessário desenvolver uma maneira de encontrar o maior elemento dentre os existentes em uma dada lista de dados.



Existem várias formas de encontrar o maior elemento de uma lista de números, de fato, este será um tópico abordado em detalhes mais adiante neste curso quando estudarmos ordenação de listas. No entanto, para fins de discussão, consideremos a versão "ingênua" abaixo:

```
Entrada:   lista L de números
Saída:     m / m > e ∀ e ∈ L

m encontrarMaior(L)
  m ← L(1)
  t ← |L|
  PARA c = 2 ATE t
    SE L(c) > m
      m ← L(c)
    FIM
FIM
```

Encontrar o maior elemento desta lista sempre demorará O(n) para cada pesquisa. Embora O(n) seja uma boa complexidade computacional, as três regras de análise de algoritmos nos impõem a seguinte pergunta:

"Podemos fazer melhor?"

Analisemos o algoritmo acima mais de perto. Adicionar um novo elemento na lista pode ser feito muito rapidamente, de fato em O(1). No entanto, se mudarmos a forma de inserir os novos elementos da lista, podemos manter o maior elemento sempre em uma posição acessível, tornando a resolução do problema de encontrar o maior elemento é checar até o último elemento da lista.

Se mudarmos a forma de inserir os novos elementos da lista podemos manter o maior elemento sempre em uma posição acessível, tornando a resolução do problema de encontrar o maior elemento factível em O(1). Note que, no entanto, a inserção dos novos elementos não terá mais complexidade O(1) mas sim, possivelmente O(n) ou ainda complexidades de mais alta ordem.

"Precisamos de um algoritmo de inserção que tenha complexidade O(n) no pior caso"

"Precisamos de uma forma de organizar os dados, uma nova ESTRUTURA DE DADOS"

Estratégias de Implementação

Filas de prioridade podem ser implementadas de diversas maneiras.

- Usando listas encadeadas como estrutura de dados base;
    - Listas encadeadas simples;
    - Listas duplamente encadeadas;

As listas encadeadas ainda podem seguir uma estratégia em que os elementos são inseridos de maneira ordenada ou não:

- Não ordenada – inserção tal qual em uma fila comum (sempre no final da fila), estratégia esta chamada de filas não ordenadas;
- Ordenada – Antes de um elementos ser inserido a fila, a posição correta (de acordo com a sua prioridade) deve ser identificada.

Na implementação não ordenada, utiliza-se uma lista encadeada como estrutura de dados base. A inserção de elementos acontece tal como na fila normal, no entanto, a remoção de itens se dá por meio da seleção do elemento de maior prioridade da fila. Para tal, toda a lista deve ser percorrida de modo a identificar o elemento com maior prioridade.

O método Inserir_com_prioridade(Fila, prioridade) deve ser composto dos seguintes passos:

- Inserir elemento no final da fila
- Atualizar a prioridade do elemento
- Ajustar ponteiros para manter a estrutura de dados consistente

O método para remoção do elemento de mais alta prioridade deve ser composto dos seguintes passos:

- Percorrer todos os elementos da lista para identificar o de maior prioridade
- Remover tal elemento da lista
- Ajustas ponteiros para manter a estrutura de dados consistente
- Retornar o elemento identificado

Uma outra possibilidade é manter a fila organizada de alguma forma para que o elemento demais alta prioridade fique sempre acessível e identificável rapidamente, fazendo com que o acesso a tal elemento seja imediato e que a sua remoção da lista tenha uma complexidade computacional baixa (idealmente O(1)).

A primeira maneira que veremos para produzir uma fila de prioridade com tais características será via a utilização de uma fila ordenada pela ordem de prioridade de seus elementos.

- Inserção O(n)
- Remoção O(1)

No final das contas, a diferença das implementações ordenada e não ordenada de filas de prioridade usando filas comuns apenas difere de em qual operação possuirá maior complexidade, se na inserção ou remoção.

Ambas as implementação solucionam o problema, no entanto, a pergunta persiste:

"podemos fazer melhor?"

A resposta é sim, Existe uma forma de implementar filas de prioridade com complexidade O(1) para acesso do elemento de maior prioridade e O(nlogn) para inserção e remoção de elementos. No entanto, devemos estudar primeiro, outra estrutura de dados chamada Heap.

Heap

A Heap é uma estrutura especializada baseada em árvores que satisfaz a seguinte propriedade:

Propriedade 1: Se B é um nó filho de A, então chave(A) ≥ chave(B).

A propriedade acima implica em que o nó raiz seja sempre o elemento com maior chave na estrutura. A estrutura acima é conhecida como **max_heap**. A propriedade 1 pode ser invertida para chave(A) ≤ chave(B) o que implica em que o nó raiz sempre terá o elemento com a menor chave. Neste caso, a estrutura será chamada de **mim_heap**.

Os principais usos da heap são na implementação eficiente de filas de prioridade, em alguns algoritmos como o Algoritmo de Djikstra, e em algoritmos de ordenação de dados tal como o algoritmo heapsort.

> ***Curiosidade***
> *A estrutura de dados heap não deve ser confundida com a heap que é comumente referenciada como a região de memória usada para a alocação dinâmica de memória. O termo foi originalmente usado para a estrutura de dados. No entanto, algumas linguagens de programação tal como Lisp implementavam a alocação dinâmica de memória usando esta estrutura de dados o que veio a conferir o nome a esta área de memória o nome.*

A heap é geralmente implementada em um array e não requer ponteiros entre os elementos.

Operações

As operações mais comuns associadas com heaps são:

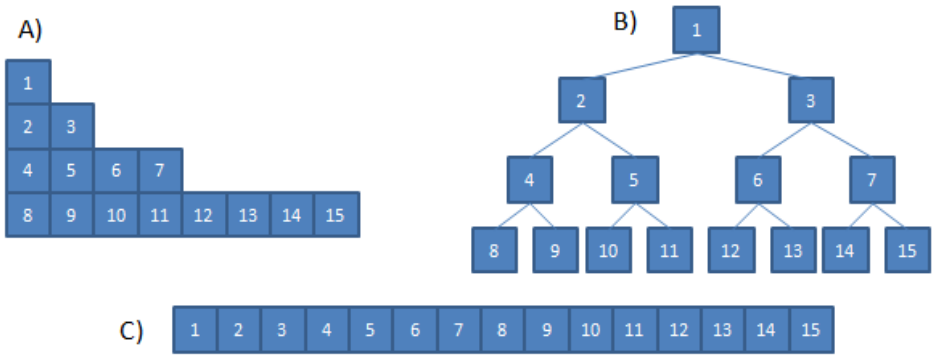
- **create_heap** - cria uma heap vazia;

- **find_max** - função que dada uma heap, encontra e retorna uma copia de seu maior elemento (Note que tal definição pressupõe uma max_heap, no caso das min_heaps a função seria melhor nomeada find_min);
- **delete_max** - remove o maior elemento da heap
- **increase_key** - incrementa a prioridade de um dado elemento. A heap deve ser re-balanceada após tal incremento;
- **decrease_key** - decrementa a prioridade de um dado elemento. A heap deve ser re-balanceada após tal decremento;
- **insert** - insere um novo elemento na heap. A heap deve ser re-balanceada após tal decremento;
- **merge** - compõe uma nova heap dadas duas heaps como entrada. A heap deve ser re-balanceada;

Heap Binária

A heap binária é um caso especial (na realidade o mais largamente utilizado) de heaps que usa como estrutura de dados base uma árvore binária. Ela possui duas propriedades:

- **Propriedade da Forma** – A árvore binária deve ser completa ou quase completa. Ou seja, todos os níveis exceto possivelmente o último estão completamente preenchidos. Se o último nível não estiver completo, os nós folha deste nível estão completamente preenchidos da esquerda para a direita;
- **Propriedade heap** – A ≥ B (no caso da max_heap) ou A≤ B (no caso da min_heap)



A primeira propriedade (propriedade da forma) impõe uma restrição quanto à forma da árvore. Tal restrição tem como razão o fato de que árvores binárias completas ou quase completas podem ser numeradas da esquerda para a direita e de cima para baixo (tal como visto no tópico de árvores binárias). Devido a tal facilidade de numeração, árvores desta forma podem ser implementadas eficientemente usando arrays como estrutura de dados base. O acesso a elementos antecessores e predecessores (pais e filhos) pode ser feito através das duas regras simples:

- Número do filho esquerdo = Número do pai x 2
- Número do filho direito = Número do pai x 2 + 1

A segunda propriedade impõe a restrição de como os elementos da árvore devem se relacionar. Tal restrição na realidade força a organização dos elementos da lista de forma que nós pais sempre possuirão chave de mais alta ordem que os filhos (o número da chave pai será sempre maior que o número da chave filho).

O que diferencia uma heap de uma árvore binária qualquer é a forma pelas quais os elementos são inseridos e removidos da heap. A idéia é inserir os elementos de tal maneira que o elemento de mais alta prioridade esteja sempre localizado na raiz, e a remoção do elemento

de mais alta ordem pode ser feita via a remoção da raiz. No entanto, tais regras de inserção e remoção demandam um trabalho adicional de modo a manter a estrutura da heap consistente.

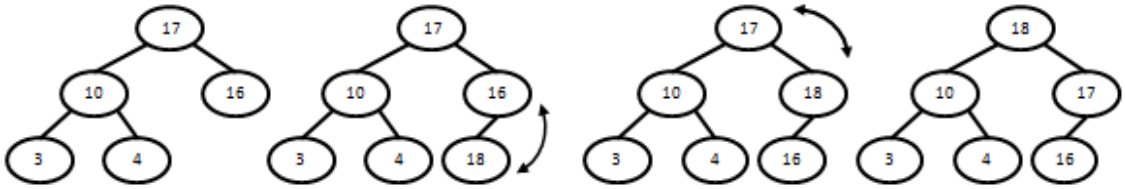
Operações básicas da heap binária

Inserção

Inserir um elemento na heap comparado com uma árvore binária requer cuidado adicional para que as duas propriedades que definem a heap sejam obedecidas. O pseudo-algoritmo de inserção é no entanto, simples:

1. Adicionar o elemento a ser inserido na próxima posição livre da heap;
2. Comparar o elemento inserido com seu pai, se eles forem iguais ou respeitarem as propriedades heap, pare;
3. Se não, troque o elemento com seu pai e retorne ao passo 2

Ex:



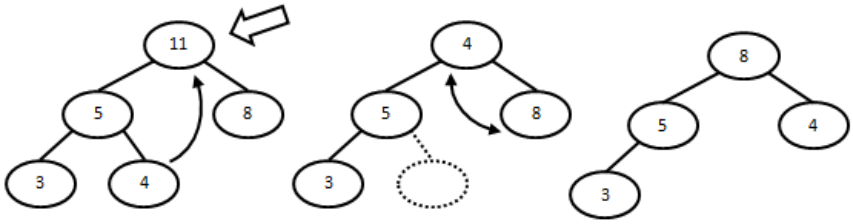
A figura acima exemplifica o processo de inserção em uma heap binária. Na primeira figura (esquerda para a direita) temos uma heap binária quase completa que respeita as duas propriedades de forma e heap. Na segunda figura inserimos o elemento 18 na primeira posição livre. Note que 18 > 16 violando a propriedade heap. Neste caso devemos trocar os elementos 16 e 18 de posição, o que pode ser visto na figura 3. Ainda na figura 3 vemos que o elemento 18 > 17 o que novamente, viola a propriedade heap. Trocamos novamente estes elementos e vemos na figura 4 que a heap agora se encontra consistente, respeitando ambas as propriedades de forma e heap.

Remoção

Identificar o elemento a ser removido em uma heap é simples. Basta removermos o nó raiz da árvore binária que implementa a heap. No entanto, após tal remoção a árvore binária ficará sem um nó raiz, de modo que um dos nós ainda existentes na árvore deve ser eleito como o novo nó raiz. O pseudo-algoritmo para a remoção do nó de maior chave na heap é dado a seguir:

1. Remover a raiz da heap e substituí-la com o último elemento do último nível;
2. Comparar a nova raiz com seus filhos, se eles respeitam a propriedade heap, parar;
3. Se não, trocar o nó raiz com um de seus filhos e retornar ao passo anterior; (menor filho na min_heap e maior filho na max_heap)

Ex:



A figura acima exemplifica o processo de remoção. A primeira figura da esquerda para a direita mostra o elemento 11 a ser removido e o elemento 4 (o último elemento alocado na heap)

que deve tomar o seu lugar. No passo seguinte comparamos o elemento 4 ao 5 e 8 e notamos que o elemento 4 deve trocar de lugar com o elemento 8. A última figura mostra a nova heap que obedece ambas as propriedades de forma e heap.

Filas de Prioridade Usando Heaps

Como discutido anteriormente, filas de prioridade podem ser implementadas de maneira ordenada ou não ordenada usando filas simples. Existe uma complexidade adicional em ordenar os elementos da fila à medida que eles são inseridos (caso ordenado) ou procurar o elemento de maior ordem no momento em que a remoção do elemento de mais alta ordem é requerida. Em ambos os casos vimos que as operações inserção ou remoção requererão O(n) e a outra O(1).

Outra forma de implementar a fila de prioridade é por meio da utilização de heaps binárias. Note que as operações de inserção e remoção ambas levam no máximo O(nlogn) o que é bom se comparado a O(n).

Exercícios

1. Implemente uma fila de prioridade não ordenada usando elementos dinâmicos (células). Proponha as estruturas de dados necessárias e implemente as funções para inserção de um novo elemento na fila e a remoção do elemento de mais alta prioridade. Utilize desenhos esquemáticos para exemplificar o funcionamento da estrutura de dados;
2. Qual a diferença entre uma fila de prioridade implementada usando listas encadeadas onde os elementos são ou não ordenados?
3. Escreva um algoritmo para a inserção de um elemento em uma heap implementada usando arrays. Proponha a estrutura de dados necessária.
4. Escreva um algoritmo para a inserção de um elemento em uma heap implementada dinamicamente. Proponha a estrutura de dados necessária.
5. Compare a dificuldade em se implementar os exercícios 3 e 4. Quais são as limitações da versão obtida no exercício 3 e as do exercício 4?
6. Repita os exercícios 3-5 para a remoção do elemento de maior prioridade.
7. Quais são as duas propriedades que uma max_heap deve apresentar? E quais são as duas da min_heap?
8. Com relação às operações increase_key(elemento) e decrease_key(elemento). Quais serão os passos necessários para garantir que a heap esteja consistente (obedecendo as duas propriedades) após a execução de cada uma destas funções. Sinta-se livre para implementar as duas funções.
9. Escreva a função heap *merge(heap *h1, heap *h2) que recebe como argumento duas heaps e retorna uma heap composta por h1 e h2. Delineie o algoritmo antes de implementá-lo.