Revisão Recursão

Resolução de problemas utilizando um caso base e um caso geral. O caso base resolve instâncias pequenas do problema. O caso geral reduz o problema em instâncias menores a serem resolvidas e chama a função recursivamente.

Definições recursivas

Alguns objetos matemáticos tem definição recursiva, como por exemplo o fatorial, palíndromos e etc. Com isso, podemos criar algoritmos recursivos que tratem essas operações.

Busca Binária

Desejamos buscar x em um vetor de números ordenado de forma crescente dados, entre as posições l e r.
Casos base:
Se o intervalo for vazio (l>r) x não está no vetor. Se dados[m] == x, onde m = (l+r)/2.

Casos gerais:
Se dados[m] < x, então x só pode estar entre m+1 e r, chamamos a recursão
Se dados[m] > x, então x só pode estar entre l e m-1, chamamos a recursão

Resumindo, recebemos um vetor ordenado e queremos encontrar um valor, percorrer o valor todo pode ser muito ineficiente. Com a busca binária, definimos um chute arbitrário para a posição média do vetor, se o elemento não foi encontrado, redefinimos os limites de busca em instâncias menores e chamamos a função recursiva até que um dos casos base sejam atingidos.

Algoritmos recursivos x iterativos

Algoritmos recursivos são geralmente mais compactos e de simples entendimento, porém em alguns casos podem ser muito ineficientes. Uma abordagem recomendada é sempre que avaliarmos um algoritmo recursivo, escrevemos o equivalente iterativo e comparamos a eficiência.

Gerenciamento de memória em algoritmos recursivos

A memória de um sistema computacional é dividida em 3 partes:
Espaço estático: contém as variáveis globais e código do programa
Heap: área que armazena a memória alocada dinamicamente
Pilha: para execução de funções

Quando uma função é invocada, suas variáveis locais são armazenadas no topo da pilha e quando ela se encerra, essas variáveis são removidas da pilha.

Em chamadas recursivas, cada chamada representa uma nova instância da mesma função na pilha.

Noções de eficiência de algoritmos

Existem formas de analisar algoritmos, independentemente do computador em que estão sendo executados (sem avaliar performance). Analisamos em função da quantidade de acessos a dados necessários, nomeando esse valor como uma variável n. Na maioria dos casos analisaremos o pior caso do algoritmo, a análise do melhor caso pode ocorrer, mas raramente é útil.

ex: algoritmo de busca sequencial simples

```
int busca(int *v, int n, int x){

        for(int i = 0; i < n; i++)
                if(v[i] == x)
                        return i;
        return -1;
}
```

consumo de tempo por linha no pior caso:

linha2: tempo c2 declaração de variável
linha3: tempo c3 atribuições, acessos e comparação, no pior caso essa linha é executada n+1 vezes (4 ciclos + teste final)
linha4: tempo c4 acessos comparação e if, no pior caso essa linha é executada n vezes
linha5: tempo c5 acesso e return
linha6: tempo c6 return

tempo de execução é menor ou igual a (pior caso): c2 + c3(n+1) + c4(n) + c5 + c6

cada ci não depende de n, depende apenas do computador, leva um tempo constante

podemos reescrever c2+c3+c5+c6+(c3+c4) x n para n >= 1
com isso vemos que o crescimento do tempo é linear em n, se n dobra, o tempo de execução praticamente dobra

definimos então uma constante d que depende apenas do computador, que possui valor estimado. Dizemos que o tempo do algoritmo é da ordem de n, a ordem de crescimento do tempo é igual a de f(n) = n. Dizemos que a função que descreve a eficiência do algoritmo é da ordem O(n)

Outro exemplo: busca binária (algoritmo nos slides)

analisamos as chamadas recursivas em função do número de posições n do vetor como na busca binárias o vetor é quebrado sempre na metade a cada chamada não bem sucedida, primeiro temos n, depois n/2, depois n/4, n/8 …, n/(2^t). O pior caso ocorre se o elemento buscado não está no vetor(l>r), quando n/2^t < 1 e n/2^t-1 >= 1, reescrevemos lg n < t < lg n + 1. Assim o número de chamadas no pior caso é t = lg n + 1

Objetivos

Temos 2 objetivos, entender o tempo de execução de um algoritmo, comparar dois algoritmos.

Nomenclatura e consumo de tempo

O(1) tempo constante: - não depende de n
- operações aritméticas
- comparações
- operadores booleanos
- acesso a posição de um vetor

O(lg n) logarítmico: - lg indica log2
- quando n dobra, o tempo aumenta em uma constante
- busca binária

O(n) linear: - quando n dobra, o tempo dobra
- busca linear

O(n lg n): - quando n dobra, o tempo um pouco mais que dobra
- alguns algoritmos de ordenação

O(n^2) quadrático: - quando n dobra, o tempo quadriplica
- BubbleSort, SelectionSort, InsertionSort

O(n^3) cúbico: - quando n dobra, o tempo octuplica
- multiplicação de matrizes n x n

Listas Ligadas

Vetores são estruturas que ocupam a memória de forma contígua(elementos ocupam uma área sequencial e definida). Isso pode gerar problemas de mal uso da memória, se alocarmos um vetor pequeno, o espaço pode ser mal utilizado e gerar sobras. Se alocarmos um vetor grande, o espaço pode não ser suficiente. Como solução podemos utilizar listas ligadas.

Listas ligadas possuem a seguinte estruturação: a memória é alocada conforme for necessário, é guardado um ponteiro para a estrutura em uma variável, cada nó da estrutura possui um ponteiro que aponta para o nó seguinte na lista, o último nó aponta para NULL. Lembrando que a memória alocada pela estrutura fica armazenada no heap e as variáveis ficam armazenadas na pilha.

Nó

Elemento de memória alocada dinamicamente, contém um conjunto de dados e um ponteiro para outro nó.

Operações

Existem várias operações que podem ser realizadas sobre listas ligadas, algumas delas são, inicialização da lista, destruição da lista, adição de elemento, remoção de elemento, impressão da lista.

Comparando desempenho entre vetores e listas ligadas

- acesso a posição k:
  vetor O(1)
  lista ligada O(k) (percorre a lista)

- inserção na posição:
  vetor O(n) (precisa mover itens para direita)
  lista ligada O(1)

- remoção da posição:
  vetor O(n) (precisa mover itens para esquerda)
  lista ligada O(1)

- uso de espaço:
  vetor provavelmente desperdiçará memória

não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

O uso de uma das opções depende do contexto do projeto

Lista Circular

Listas circulares são listas ligadas nas quais a extremidade final da liga se liga novamente no primeiro elemento. Podem ser usadas na execução de processos no sistema operacional.

Lista circular com nó cabeça

Listas circulares com nó cabeça, possuem um nó inicial chamado de dummy, esse nó não armazena dados da lista, mas pode ser usado para armazenar informações gerais sobre a estrutura. Facilita código de inserção e remoção, ao percorrer a cabeça deve ser ignorada.

Lista duplamente ligada

Listas duplamente ligadas, possuem ponteiros para o próximo elemento na lista e para o elemento anterior na lista, possui ponteiro para marcar o início e o fim da lista. Pode existir uma lista duplamente ligada e circular, com os nós para próximo e anterior das extremidades se ligando.

Filas e Pilhas

Filas

Filas são estruturas que seguem o padrão FIFO(first-in first-out), no qual o primeiro elemento a ser inserido é o primeiro a ser retirado. Possui 2 operações, enfileira(queue) que adiciona um elemento no fim e desenfileira(dequeue) que remove um elemento do início. Filas podem ser implementadas com lista ligada, guardando um ponteiro para o início e um para o fim da fila.

Implementação com lista ligada

Para enfileirar elementos, alteramos o ponteiro de elemento seguinte do último elemento da fila, fazendo ele apontar para o novo elemento, em seguida atualizamos o ponteiro que marca o fim para o novo elemento no fim da fila, o ponteiro para elemento seguinte do novo fim é setado para NULL.

Para desenfileirar elementos, utilizamos um ponteiro auxiliar para guardar o ponteiro para o primeiro elemento, retiramos o dado a ser usado, definimos o novo inicio para o proximo elemento do inicio e liberamos o auxiliar que aponta para o antigo primeiro.

Outra opção utilizando lista ligada é criar uma lista circular com nó cabeça, com um ponteiro apontando para o último elemento da fila. Para enfileirar fazemos o próximo elemento do fim apontar para o novo elemento, atualizamos o ponteiro que aponta para o fim e fazemos o novo nó apontar para o nó cabeça. Para desenfileirar removemos o nó seguinte ao nó cabeça.

Implementação com vetor

Inserir no final do vetor: O(1)
remover do começo do vetor: O(n) (é necessário arrumar o vetor)

Podemos utilizar 2 variáveis ini e fim para marcar o início e fim da fila dentro do vetor, porém é necessário reorganizar o vetor constantemente, o que leva tempo O(n).

Para solucionar esse problema, utilizamos um vetor circular. Armazenamos variáveis para guardar o inicio da fila, o fim da fila, o tamanho total disponível e o tamanho atualmente ocupado. Para enfileirar, adicionamos o elemento na posição que marca o fim e atualizamos o novo fim para (novo_fim + 1) % tamanho atual (módulo), a operação de módulo garante o reinício do ciclo quando a posição ultrapassa o tamanho máximo do vetor. A mesma lógica é utilizada para desenfileirar.

A implementação com vetor é útil quando a fila não é tão grande, pois o gasto de desempenho não é tão alto e há a economia ao não utilizar tantos ponteiros. Porém se o número de elementos é alto, é mais efetivo a utilização de lista ligada.

Exemplos de aplicações de filas

Gerenciamento de fila de impressão
Buffer do teclado
Escalonamento de processos
Comunicação entre aplicativos/computadores
Percurso de estruturas de dados complexas(grafos etc).

Pilha
Pilhas são estruturas que seguem o padrão LIFO (last-in first-out), no qual o último elemento adicionado é o primeiro a ser retirado.

Operações sobre pilhas

Empilha(push): adiciona no topo da pilha
Desempilha(pop): remove do topo da pilha

Implementação com vetor

Na implementação com vetor utilizamos uma variável para marcar a posição de topo da pilha. A função de empilhar insere um elemento e incrementa o topo. A função de desempilhar decrementa o topo e devolve o elemento nessa posição.

Implementação com lista ligada

Com lista ligada, armazenamos um ponteiro para o topo da pilha. A função empilhar faz o ponteiro de próximo elemento do novo elemento apontar para o topo da pilha e atualiza o topo para o novo elemento. A função desempilhar usa um auxiliar para armazenar o topo da pilha, extrai o dado necessário, atualiza o topo para o próximo elemento do topo atual e libera a memória do auxiliar.

Exemplos de aplicações de pilha

Balanceamento de parênteses(expressões matemáticas, HTML…)
Cálculo e conversão de notações(pré-fixa, pós-fixa, infixa)
Percurso de estruturas de dados complexas(grafos etc)
Recursão

Árvores Binárias

Árvores binárias são estruturas que definem uma hierarquia de dados. Uma árvore é composta por nós, esses nós possuem filhos que podem possuir filhos e assim por diante. O nó do topo da árvore é chamado de nó raíz. Os nós terminais da árvore, que não possuem nenhum filho, são chamados de nós folha.

Cada nó possui pelo menos 2 filhos, um da esquerda e um da direita. A árvore formada a partir do primeiro nó a direita do nó raiz é chamada de subárvore direita, assim como a sua oposta do lado esquerdo é chamada de subárvore esquerda.

Árvores binárias possuem níveis de altura. A altura de uma árvore é definida pelos nós em níveis diferentes mais o nó raiz.

Relação entre altura e número de nós

Se uma árvore tem altura h, então a árvore tem no mínimo h nós e no máximo 2^h - 1 nós.

Se a árvore tem n>=1 nós então, a altura é no mínimo [lg(n+1)] e no máximo n

Implementação com lista ligada

Podemos implementar uma árvore binária com lista ligada. Cada nó da árvore possui um ponteiro para um filho a esquerda e um filho a direita. Os nós folha apontam seus dois ponteiros para NULL. Também é possível implementar a versão com um ponteiro que aponta para o nó pai.

Operações sobre árvores binárias

A maioria das operações que percorrem as árvores binárias são recursivas. Elas utilizam o conceito de descer nas subárvores a esquerda e a direita com chamadas recursivas. As principais operações que percorrem a árvore são para busca e impressão dos dados. Existem 4 tipos diferentes que podemos utilizar para percorrer uma árvore binária:

Percurso pré-ordem: primeiro visita e processa a raiz, depois a subárvore esquerda, depois a subárvore direita.

Percurso pós-ordem: primeiro visita a subárvore esquerda, depois a subárvore direita e depois a raiz.

Percurso inordem: primeiro visita a subárvore esquerda, depois a raiz e depois a subárvore direita.

Percurso em largura: visita os nós por níveis, da esquerda para a direita. O percurso em largura é implementado utilizando uma fila, colocamos a raiz na fila e a cada elemento desenfileirado, seus 2 filhos são adicionados na fila.

Árvores binárias de busca

Árvores binárias de busca são estruturas de árvore implementadas com o objetivo de conseguir otimização na busca por dados. Nessa estrutura, cada nó da árvore contém um elemento de um conjunto ordenável. Cada nó r com subárvores esquerda Te e direita Td satisfaz as seguintes propriedades:

- e < r para todo elemento e em Te, ou seja, todos os elementos a esquerda do nó devem ser menores que ele
- d > r para todo elemento d em Td, ou seja, todos os elementos a direita do nó devem ser maiores que ele

Busca por um valor

A idéia é semelhante a busca binária:

- o valor buscado está na raiz da árvore
- ou é menor que o valor da raiz, nesse caso buscamos na subárvore esquerda

- ou é maior que o valor da raiz, nesse caso buscamos na subárvore direita

A eficiência da busca depende da forma da árvore, em uma árvore com n elementos adicionados aleatoriamente, a busca demora em média O(lg n).

Inserindo um valor

Para inserir elementos, percorremos a árvore testando as subárvores esquerdas e direitas, assim como na busca. Quando encontrarmos a posição correta que possui NULL, inserimos o nó atualizando os ponteiros.

Mínimo da árvore

O valor mínimo de uma árvore binária de busca é o menor valor na subárvore esquerda. Encontramos realizando o mesmo procedimento de percorrer e realizando os testes.

Máximo da árvore

O valor máximo da árvore será o maior valor na subárvore direita, utilizamos o mesmo padrão anterior para encontrá-lo.

Sucessor

Dado um nó da árvore, podemos encontrar seu sucessor. O sucessor de um nó é o valor mínimo de sua subárvore direita. Caso o nó não tenha subárvore direita, seu sucessor é seu ancestral a direita ou ele é o maior nó e não possui sucessor.

Antecessor

Operação simétrica de sucessor.

Remoção

A operação de remoção percorre a árvore até encontrar o elemento e atualiza os ponteiros da forma correta para desligar o nó da árvore. Após isso, o nó é liberado. O nó a ser substituído no lugar do nó a ser removido, é o nó sucessor do nó removido.

Árvores balanceadas

Árvores balanceadas são estruturas que fixam uma eficiência para operações sobre a árvore, deixando de depender da altura como na árvore binária de busca. Não são

o melhor tipo de árvore possível mas são “quase”. As operações tem eficiência O(lg n).

Árvores Rubro-Negras Esquerdistas

Uma árvore rubro-negra esquerdista é uma árvore binária de busca tal que:

- todo nó é ou vermelho ou preto
- a raiz é preta
- as folhas são NULL e tem cor preta
- se um nó é vermelho, seus dois filhos são pretos, ele é o filho esquerdo do seu pai(por isso esquerdista)
- em cada nó, todo caminho dele para uma de suas folhas descendentes tem a mesma quantidade de nós pretos, não contamos o nó e chamamos de a altura-negra do nó

Altura da árvore rubro-negra esquerdista

Seja bh a altura-negra da árvore, a árvore tem pelo menos 2^bh - 1 nós não nulos:

Utilizando indução:

se bh = 0:

- a árvore é apenas uma folha NULL
- tem exatamente 2^bh - 1 = 0 nós não nulos

se bh > 0:

- seus filhos tem altura-negra pelo menos bh-1
- cada subárvore tem pelo menos (2^bh-1) - 1 nós não nulos
- a árvore tem pelo menos 2((2^bh-1)-1)+1 nós não nulos
- ou seja, tem pelo menos 2^bh - 1 nós não nulos

A altura-negra bh é pelo menos metade da altura h da árvore
Não existe nó vermelho com filho vermelho
o número de nós não nulos n é n >= 2^bh-1 >= 2^h/2 - 1
ou seja h <= 2lg(n+1) = O(lg n)

Implementação e operações nos slides.

Fila de prioridades e Heap

Uma fila de prioridades é uma estrutura de dados com duas operações básicas, inserir um novo elemento e remover o elemento com maior prioridade.

Uma pilha é como uma fila de prioridades, o elemento com maior prioridade é sempre o último a ser inserido.

Uma fila é como uma fila de prioridades, o elemento com maior prioridade é sempre o primeiro a ser inserido.

Implementação da fila de prioridades com vetor

Para realizar as operações de uma fila de prioridades, primeiro escrevemos uma função troca, que troca duas posições do vetor, isso será uma prática muito usada pela fila. As operações básicas de uma fila de prioridade são:

inserir: para inserir utilizamos uma variável que marca a próxima posição livre na fila e inserimos nela, atualizando a variável.

extrair máximo: nessa função, utilizamos um for para filtrar a posição em que está o elemento de prioridade máxima, utilizamos a função de troca para trocar o elemento nessa posição pelo elemento na última posição da fila. Decrementamos o número de elementos e retornamos o elemento na posição n que é o elemento extraído.

com os elementos dispostos aleatoriamente a inserção leva tempo O(1) e a extração O(n)

já mantendo o vetor ordenado, a inserção leva O(n) (necessidade de ordenar) e a extração leva O(1) (não precisa buscar máximo, está na primeira ou na última posição, dependendo do tipo de ordenação).

Árvores binárias completas

Uma árvore binária é completa se todos os níveis exceto o último estão cheios(todo nó tem 2 filhos) e os nós do último nível estão o mais a esquerda possível.
Esse tipo de árvore pode ser representado utilizando vetor, no qual cada nó ocupa uma posição sequencial no vetor descendo nos niveis da árvore da esquerda para a direita. Não precisamos de ponteiros.

O filho esquerdo de um nó na posição i do vetor está na posição 2*i+1 e o filho direito na posição 2*i+2. O pai do nó i está na posição (i-1)/2.

Heap de máximo

Um heap de máximo é uma árvore binária completa, no qual os filhos são menores ou iguais ao pai, ou seja, a raiz é o máximo(não é uma árvore binária de busca).

Inserindo no heap de máximo:

- para inserir, subimos no heap trocando o pai se necessário, até levar o valor para sua posição correta, para isso utilizamos o heap como uma fila de prioridades

Extraindo o máximo:

- trocamos a raiz com o último elemento do heap
- descemos no heap arrumando
- atualizamos o tamanho do heap

Mudando prioridade de um item:

- se a prioridade aumenta, subimos no heap arrumando
- se a prioridade diminui, descemos no heap arrumando

Esse tipo de percurso necessita saber a posição do item no heap e para encontrá-lo pode levar O(n). Como solução, podemos atribuir ids de 0 a n-1 aos itens. Criamos um vetor de tamanho n que armazena a posição do item no heap, em O(1) encontramos a posição do item no heap. Toda vez que o heap for atualizado, o vetor de posições deve ser atualizado também.

—----------

Algoritmos de ordenação

Veremos algoritmos de ordenação de elementos em um vetor/lista.

BubbleSort

A ideia do BubbleSort é ordenar um vetor do fim para o começo, trocando pares invertidos. Por exemplo, para uma ordenação crescente, percorremos o vetor com dois índices, podemos chamar de i e j. O índice i percorre o vetor de 0 até i menor que n-1(até o penúltimo elemento) incrementando i. Enquanto o índice j percorre o vetor de n-1 até j maior que i, decrementando j. Implementamos isso com 2 laços for aninhados e testamos no for mais interno, se vetor[j] < vetor[j-1] então trocamos vetor[j-1] por vetor[j]. Resumindo, o laço interno percorrerá todos os membros do vetor, iniciando no último, testando se o elemento anterior é maior que o elemento atual, se for, realiza a troca. Isso se repetirá n vezes, diminuindo o limite do laço interno pelo laço externo e ordenará o vetor. Podemos definir uma variável que marca se houveram trocas, definimos ela como falsa e entramos no laço interno, se houver alguma troca a marcamos como verdadeira. Na verificação do laço externo verificamos a variável, se não houver trocas é porque o algoritmo terminou e então definimos um ponto de parada.

No pior caso, o BubbleSort possui eficiência de $O(n^2)$ para comparações e trocas, a segunda versão com variável troca é um pouco mais eficiente.

Ordenação por inserção (InsertionSort)

No insertion sort a ideia é de que se já tivermos v[0], v[1], …, v[i-1] ordenado, então inserimos v[i] na ´posição correta. Muito parecido com o BubbleSort, porém dessa

vez testamos os índices começando pelo segundo elemento com seu elemento anterior, trocamos de acordo com a política de ordenação desejada. Novamente, podemos testar se o elemento já está na posição correta, não será mais necessário continuar percorrendo testando.

No pior caso, o InsertionSort também possui eficiência $O(n^2)$ para comparações e atribuições.

Ordenação por seleção (SelectionSort)

A ideia do algoritmo de ordenação por seleção em ordem crescente, é percorrer os elementos do vetor guardando a posição do valor mínimo. Percorremos o vetor com um índice em um for interno buscando o valor mínimo, após o término do laço interno, trocamos o minimo com a primeira posição do vetor marcada por um índice i em um for externo, repetimos os passos até o fim do vetor.

No pior caso, o SelectionSort tem eficiência de $O(n^2)$ nas comparações e de $O(n)$ para trocas.

Comparação entre os três algoritmos

Comparando o BubbleSort otimizado, o InsertionSort otimizado e o SelectionSort, obtemos que o InsertionSort otimizado é o mais eficiente, seguido pelo SelectionSort e por último o BubbleSort otimizado. Para um n = 30000, o SelectionSort leva 4,2 vezes o tempo do InsertionSort, enquanto o BubbleSort leva 9,3 vezes o tempo do InsertionSort.

Ordenação utilizando fila de prioridades (HeapSort)

Podemos utilizar uma fila de prioridades para ordenar um vetor. Copiamos todos os elementos do vetor para a fila de prioridades, utilizamos a operação de extrair valor com chave máxima para atribuir os elementos ordenados de volta ao vetor, utilizando o próprio valor do elemento como chave. Essa forma de ordenação possui eficiência de $O(n \lg n)$, porém perdemos tempo para copiar o vetor para o heap(fila de prioridades). Não precisamos perder esse tempo, podemos utilizar o método para transformar um vetor em um heap, chamamos esse algoritmo de HeapSort(ver slides).

Conclusão

Vimos três algoritmos com eficiência $O(n^2)$. O BubbleSort na prática é o pior dos três e raramente é utilizado. o SelectionSort é bom quando comparações são muito mais

baratas que trocas. O InsertionSort é o melhor dos três na prática. Vimos um algoritmo melhor assintoticamente que é o HeapSort.

Algoritmos de ordenação, divisão e conquista

Algoritmos de divisão e conquista utilizam a estratégia de dividir um conjunto de dados em 2 partes e ordená-las separadamente, para depois ordenar o conjunto todo através da intercalação dos vetores em um vetor auxiliar. Podemos usar recursão para implementar esses algoritmos, já que a recursão parte do princípio de que é mais fácil resolver problemas menores.

MergeSort, ordenação por intercalação
Intercalação
Dado um vetor v, definimos dois limites que separam o vetor em 2 partes, um da posição L até M e outro da posição M+1 até R. Utilizamos um vetor auxiliar do tamanho do vetor original. Temos as variáveis i = L, j = M+1, k = 0 e r

Implementamos a função merge. Essa função primeiro intercala os valores, percorrendo o vetor original da posição i e da posição j, compara os valores e armazena o menor deles(caso de ordenação crescente) na posição k do vetor auxiliar, atualizando os indices i, j e k, até que um dos subvetores chegue ao fim. Depois implementamos os laços que copiam o resto do subvetor de indice i ou do subvetor de indice j no vetor auxiliar. Por último, copiamos todos elementos do auxiliar de volta no vetor original.

Ordenação
Dividimos um vetor de tamanho n em dois, marcando a posição de inicio l e a posição final r, guardando uma variável m = (l+r)/2, dividindo o vetor na metade em dois subvetores. Chamamos a função recursiva, dividindo o primeiro subvetor, que guarda a metade inferior do vetor, em metades até o subvetor atingir o caso base que é um vetor de tamanho 0 ou 1. Depois realizamos nova chamada recursiva para o segundo subvetor, que guarda a metade superior do vetor, realizando o mesmo procedimento. Ao final, chamamos a função merge, que irá intercalar os elementos conforme a pilha da chamada recursiva termina.

Para um n qualquer, a eficiência do MergeSort é de O(n lg n).

QuickSort
No quicksort, primeiro escolhemos um pivô (geralmente a posição do meio do vetor) e colocamos os elementos menores que o pivô na esquerda e os maiores que o pivô na direita, o pivô já está na posição correta. Ordenamos o lado esquerdo e direito como subvetores.

Partição
Implementamos a função partition responsável por dividir o vetor em menores e maiores que o pivô e retornar a posição do pivô. Percorremos o vetor com um índice i que começa na última posição e um índice r que começa uma posição acima da última posição(fora do vetor). Andamos em um laço que inicia na última posição do vetor e vai até i >= 1, decrementando i. testamos se o elemento da posição i é maior ou igual ao pivô, se sim, ele está no lado correto(lado direito), decrementamos o índice r e trocamos o elemento na posição i com o elemento na posição r.

Ordenação
Seguindo o mesmo princípio recursivo do MergeSort, realizamos o particionamento do vetor com chamadas recursivas nas duas metades, maiores e iguais ao pivô e menores que o pivô.

No pior caso, o QuickSort tem eficiência de $O(n^2)$, porém no caso médio tem eficiência O(n lg n), se equiparando ao MergeSort.

Tabela de espalhamento - Hash

A ideia de uma tabela hash é organizar um conjunto de dados em posições de uma tabela, mediante a uma chave que irá gerar a posição em que esse dado será inserido. Ex: queremos organizar palavras, podemos gerar uma chave utilizando características de cada palavra e mapear sua posição na tabela. Uma tabela de espalhamento é um TAD(tipo abstrato de dado - conjunto de dados e operações que podem ser realizadas sobre eles) para conjuntos dinâmicos com certas propriedades:

- os dados são acessados por meio de um vetor de tamanho conhecido
- a posição do vetor é calculada por uma função de hashing

Geralmente tabelas de hash são implementadas com um vetor e as linhas da tabela são listas ligadas.
Características das tabelas de espalhamento

Restrições:

- estimativa do tamanho do conjunto de dados deve ser conhecida
- bits da chave devem estar disponíveis

Tempo das operações:

- depende da função de hashing
- chaves bem espalhadas possuem tempo “quase” O(1)
- se temos n itens e uma tabela de tamanho M, o tempo de acesso é o tempo de calcular a função de hashing mais O(n/M)
- para chaves muito agrupadas, que representam o pior cenário de tempo, a eficiência é de O(n)

Funções de Hashing

Uma boa função de hashing deve espalhar bem. A probabilidade de uma chave ter um hash específico é de aproximadamente 1/M. Esperamos que cada lista tenha n/M elementos, ou seja, os dados sejam divididos igualmente entre as linhas da tabela.
Existem métodos genéricos que funcionam bem na prática:

- método da divisão
- método da multiplicação

Se conhecermos todas as chaves posteriormente, é possível encontrar uma função de hashing injetora, ou seja, que não gera colisões. Essas funções são difíceis de encontrar.

Interpretando chaves

Pressupomos que as chaves são números inteiros. Se não forem, interpretamos como uma sequência de bits. Ex: a palavra “bala”, cada letra pode ser reinterpretada como seu equivalente como cadeia de bits. Porém esse método gera um número alto que pode explodir rapidamente, veremos como contornar esse problema.

Método da divisão

Obtemos o resto da divisão inteira pelo tamanho M do hashing:

$h(x) = x \bmod M$

ex: h(“bala”) = 1.650.551.905 mod 1783 = 277

Escolher M como uma potência de 2 não é uma boa ideia, pois considera apenas os bits menos significativos. Normalmente escolhemos M como um número primo, longe de uma potência de 2.

Método da divisão para strings

Queremos calcular o número x que representa “bala”:

$x = \text{‘b’}.256^3 + \text{‘a’}.256^2 + \text{‘l’}.256^1 + \text{‘a’}.256\text{^}0$
$= (((\text{‘b’}).256+\text{‘a’}).256+\text{‘l’}).256+\text{‘a’}$

porém x mod M pode estourar um int rapidamente

utilizamos: (((‘b’ mod M).256+’a’ mod M).256 + ‘l’ mod M).256+ ‘a’ mod M

Método da multiplicação

Multiplicamos por um certo valor real A conveniente e obtemos a parte fracionária. A posição relativa do vetor não depende M:

h(x) = [M (A . x mod 1)]

exemplo:

$$\begin{aligned} h(\text{“bala”}) &= \lfloor 1024 \cdot [((\sqrt{5}-1)/2 \cdot 1.650.551.905) \bmod 1] \rfloor \\ &= \lfloor 1024 \cdot [1020097177{,}4858876 \bmod 1] \rfloor \\ &= \lfloor 1024 \cdot 0{,}4858876 \rfloor \\ &= \lfloor 497{,}5489024 \rfloor = 497 \end{aligned}$$

O uso da razão áurea como valor de A é sugestão de Knuth

Segurança de programas que usam hashing

Sabendo a função de hashing, podemos prejudicar o programa, inserindo muitos elementos com o mesmo hash. Para resolver esse problema, podemos escolher uma função de hashing aleatória. Para escolher uma boa função de hashing aleatória:

- fixe p um primo maior do que M
- escolha um a entre 1 e p e b entre 0 e p ao acaso
- defina h(k) = ((ak + b) mod p) mod M

sabemos que essa função espalha bem e a probabilidade de colisão é no máximo 1/M, portanto é um hashing universal.

Endereçamento aberto

Alternativa a tabela hash com lista ligada. Os dados são guardados no próprio vetor, colisões são colocadas em posições livres da tabela.

Características:

- evita percorrer usando ponteiros e alocação de memória
- se a tabela encher, deve recriar uma tabela maior e mudar a função de hashing
- remoção é mais complicada

Endereçamento aberto com sondagem linear

Inserção:

- geramos a posição com a função de hash e se não houver ocupada, inserimos
- se houver colisão, procuramos a próxima posição livre (módulo M)

Busca:

- simulamos a inserção
- calculamos a função de hashing, procuramos na tabela
- se encontrarmos, devolvemos o elemento, se encontrar uma posição vazia, retorna NULL

  Em um vetor de ponteiros, o espaço vazio é representado por NULL. Em um vetor comum, devemos setar um espaço vazio com um valor, chamamos de dummy, geralmente utilizamos valores que nunca serão usados como dados ou temos um campo indicando dummy

Remoção:

- apenas remover os elementos da tabela, irá quebrar a busca, já que ela para ao identificar posições vazias
- temos 3 opções:
  - rehash: removemos os elementos até a próxima posição vazia após o elemento removido, recalculamos o hash de cada um e reinserimos na tabela. Pode ser custoso e difícil de implementar
  - troca por valor dummy: definimos um dummy que indica que o valor foi removido. Não pode ser o mesmo que indica espaço vazio.
  - marcação como removido utilizando campo adicional

  Caso a remoção seja feita utilizando marcação de item removido, devemos adaptar a inserção e busca para essa mudança.

Hashing Duplo

É como a sondagem linear, com uma estratégia mais geral para lidar com conflitos. Ao invés de saltarmos de 1 em 1 na inserção, saltamos seguindo uma segunda função de hashing:

$h(k,i) = (hash1(k) + i.hash2(k)) \bmod M$

- hash2 nunca pode ser zero
- hash2 precisa ser coprimo com M

exemplos: - escolha M como uma potência de 2 e faça hash2 seja sempre ímpar

- escolha M como um número primo e faça que hash2 < M

Eficiência dos hashings

Sondagem linear - número de acessos médio por busca

n/M	$1/2$	$2/3$	$3/4$	$9/10$
com sucesso	1.5	2.0	3.0	5.5
sem sucesso	2.5	5.0	8.5	55.5

Hashing duplo - número de acessos médio por busca

n/M	$1/2$	$2/3$	$3/4$	$9/10$
com sucesso	1.4	1.6	1.8	2.6
sem sucesso	1.5	2.0	3.0	5.5

De qualquer forma, é muito importante não deixar a tabela encher muito:

- Você pode aumentar o tamanho da tabela dinamicamente
- Porém, precisa fazer um rehash de cada elemento para a nova tabela

[1]Baseado em Sedgewick, R. Algorithms in C, third edition, Addison-Wesley. 1998.

26

Conclusão

Hashing é uma boa estrutura de dados para:

- inserir, remover e buscar dados pela sua chave rapidamente
- com uma boa função de hashing, temos eficiência de O(1)
- não é boa se quisermos fazer operações relacionadas a ordem das chaves

Escolhendo implementação:

- Sondagem linear é o mais rápido se a tabela for esparsa
- Hashing duplo usa melhor a memória, porém gasta mais tempo para computar duas funções de hashing
- Encadeamento separado é mais fácil de implementar, porém usa memória a mais para os ponteiros

Além disso, funções de hashing tem várias outras aplicações:

- Para evitar erros de transmissão, podemos, além de informar uma chave, transmitir o resultado da função de hashing: dígitos verificadores, sequências de verificação para arquivos (MD5 E SHA)
- Guardamos o hash de uma senha no banco de dados ao invés da senha em si: evitamos vazamento de dados em caso de ataque, mas a

probabilidade de duas senhas possuírem o mesmo hash deve ser mínima