MC202 - Estruturas de Dados

Guilherme P. Telles

IC

5 de Novembro de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides foram escritos usando português anterior à reforma ortográfica de 2009.

Conjuntos

Conjunto

- Um conjunto é uma coleção de elementos distintos de um conjunto universo U. A ordem relativa entre os elementos não é importante.
- As operações típicas são:
  - inserir um elemento em S,
  - remover um elemento de S,
  - testar se um elemento pertence a S,
  - calcular união, interseção e diferença de dois conjuntos.

Implementação

- Vamos supor que os elementos de U não formam um intervalo denso (não são $\{1, \ldots, |U|\}$ nem podem ser mapeados trivialmente em $\{1, \ldots, |U|\}$).
- Um vetor ordenado, uma lista, uma tabela de hashing ou árvore de busca balanceada de elementos de U podem ser usados para implementar um conjunto, dependendo das operações mais freqüentes.

Elementos de um intervalo denso

- Vamos supor que os elementos de U formam um intervalo denso (são $\{1, \ldots, |U|\}$ ou podem ser mapeados trivialmente em $\{1, \ldots, |U|\}$).
- Nesse caso podemos usar um vetor de $|U|$ bits para representar um subconjunto.
- As operações serão rápidas.
- Pode ser econômica em memória, principalmente para subconjuntos densos.

Mapas

Map

- Um *map* é uma coleção de pares ordenados $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ tais que os x são distintos. Um map é visto como uma função do domínio definido pelo conjunto dos x na imagem definida pelo conjunto dos y.
- As operações típicas são:
  - inserir um par (x, y),
  - recuperar y dado x,
  - atualizar y dado x,
  - remover um par dado x.

Implementação

- Vamos supor que os elementos do domínio formam um intervalo denso (são $\{1, \ldots, n\}$ ou podem ser mapeados trivialmente em $\{1, \ldots, n\}$).
- Nesse caso podemos usar um vetor para representar um map.
- As operações serão rápidas e o uso de memória tenderá a ser ótimo.

- Se os elementos do domínio não formam um intervalo denso, podemos armazenar o map em um vetor ordenado, lista auto-organizável, tabela de hashing ou árvode de busca binária balanceada, dependendo das operações mais freqüentes.

Filas de prioridades

Fila de prioridades

- Uma fila de prioridades é uma estrutura de dados para armazenar chaves com prioridade, com as operações básicas:
  - Inserir uma chave.
  - Recuperar a chave com prioridade máxima.
  - Remover a chave com prioridade máxima.
- Também estamos interessados nas operações
  - Aumentar a prioridade de uma chave.
  - Remover uma chave.
- Também pode ser definida para prioridade mínima.

Implementação

- Várias estruturas de dados podem ser usadas, como:
  - vetor,
  - vetor ordenado,
  - árvore binária de busca,
  - árvore binária de busca balanceada,
  - heap de máximo
  - etc.
- A comparação entre essas soluções é um exercício na lista.

Heaps

Heap binário

- Um *heap binário* é uma árvore binária em que cada nó armazena uma chave e que satisfaz a seguinte propriedade:

  a chave em qualquer nó v é maior ou igual às chaves nos filhos dele.

- Dessa forma, a chave em um nó v é a maior chave no sub-heap enraizado em v.
- Esse heap é chamado de *heap de máximo*.

- A propriedade do heap pode ser definida também para que a raiz do heap seja a chave mínima no heap.
- Esse heap é chamado de heap de mínimo.
- Todos os algoritmos para o heap de máximo se aplicam ao heap de mínimo.

- Cada nó do heap pode armazenar um par *(chave,prioridade)* e as propriedades podem ser definidas sobre as prioridades.
- Vamos supor que temos apenas chaves. Nada significativo muda se tivermos prioridades também.

Representação

- Um heap pode ser mantido na forma de uma árvore binária completa.
- Sendo uma árvore completa, ele pode ser representado de forma implícita, usando pouca memória.
- (As chaves são colocadas em um vetor H. A raiz está em $H[0]$ e os filhos do nó na posição i estão nas posições $2i+1$ e $2i+2$. O pai de um nó $H[i]$ está em $\lfloor \frac{i-1}{2} \rfloor$.)
- As operações de inserção, remoção de máximo e atualização de chave fazem trabalho proporcional a $\log_2 n$.
- Isso faz do heap uma ótima alternativa para implementar uma fila de prioridades.

Inserção

- A nova chave é inserida em $H[n]$ e movida em direção à raiz até se tornar maior que as chaves dos filhos.

Inserção

- Seja H um heap com n nós. $H[0, n-1]$ é um heap válido.
- A nova chave é adicionada em $H[n]$. $H[n]$ é um heap unitário e é válido.
- Se $H[n] < H[\lfloor \frac{n}{2} \rfloor]$ então $H[0, n]$ é um heap válido e a inserção termina.
- Senão, $H[n] > H[\lfloor \frac{n}{2} \rfloor]$ e essas chaves são trocadas. A subárvore H' enraizada em $H[\lfloor \frac{n}{2} \rfloor]$ passa a ser um heap válido. A subárvore $H[0, n-1] \setminus H'$ também é um heap válido.
- O mesmo procedimento é aplicado novamente a $H[\lfloor \frac{n}{2} \rfloor]$.

Remoção do máximo

- A chave em $H[n-1]$ é colocada em $H[0]$. Depois essa chave é movida em direção a uma folha até se tornar maior que as chaves dos filhos.

Remoção do máximo

- Seja H um heap com n nós.
- $H[1, n-1]$ contém dois heaps válidos com raízes $H[1]$ e $H[2]$.
- $H[n-1]$ é movido para $H[0]$ e o tamanho do heap é reduzido de uma unidade.
- A chave em $H[0]$ é trocada com o máximo dentre $H[1]$ e $H[2]$.
- Suponha que $H[1]$ seja o máximo. Então o heap enraizado em $H[2]$ continua válido. Depois da troca $H[0]$ tem a maior chave no heap.
- O mesmo procedimento é aplicado ao heap enraizado em $H[1]$.

Número de operações

- O número de operações para uma troca de chaves é constante para cada nó ao longo de um caminho da raiz até uma folha ou vice-versa.
- O tamanho desse caminho é limitado pela altura h do heap. Então o número de operações realizadas durante a inserção ou da remoção é no máximo

$$ch = c\lfloor \log_2 n \rfloor,$$

para uma constante c.

Aumentar o valor de uma chave

- Para encontrar uma chave no heap é preciso percorrê-lo inteiro.
- Para melhorar a eficiência podemos usar uma outra estrutura para indexar os nós do heap. Por exemplo,
  - Vetor: se as chaves são distintas e estão em um intervalo conhecido e de tamanho razoável, podemos usar um vetor com uma posição para cada chave.
  - Tabela de hashing: tende a degradar com muitas atualizações.
  - Árvore balanceada: vai permitir operações em tempo proporcional a $\log_2 n$, mas usa mais memória.
- Depois de encontrar a chave, basta mover para cima.

Construção a partir de um vetor

- Dado um vetor H com n chaves, podemos construir um heap de duas formas:
  - top-down
  - bottom-up

Construção top-down

- Aplicamos a inserção $n-1$ vezes: $H[0,0]$ é um heap e $H[1], H[2], \ldots, H[n-1]$ são inseridos no heap seqüencialmente.

Trabalho

- No pior cenário possível, cada chave que é adicionada faz um caminho de uma folha até a raiz.
- A soma dos tamanhos de todos os caminhos dos nós até a raiz é proporcional a $n \log_2 n$.
- Como o trabalho é constante para cada nó ao longo de um caminho, então o trabalho total da construção top-down é proporcional a $n \log_2 n$.

Construção bottom-up

- Vamos supor que n é da forma $2^{h+1} - 1$, isto é, o heap é cheio.
- Todas as folhas, $H[2^h, 2^{h+1} - 1]$ são heaps unitários.
- Para cada par de folhas irmãs, se o pai for menor que uma delas então trocamos ele e a maior, obtendo um heap válido de tamanho 3.
- Continuamos ao longo dos níveis da árvore, obtendo heaps de tamanhos $7, 15, 31, \ldots$

Trabalho

- No pior cenário, sempre trocamos um pai com os descendentes, até alcançar uma folha.
- No nível h, nenhuma das $\frac{n}{2}$ chaves é trocada, no nível $h-1$ $\frac{n}{4}$ chaves são trocadas 1 vez, no nível $h-2$ $\frac{n}{8}$ chaves são trocadas 2 vezes, no nível $h-3$ $\frac{n}{16}$ chaves são trocadas 3 vezes e assim por diante.
- Então o número de trocas é $1\frac{n}{4} + 2\frac{n}{8} + 3\frac{n}{16} + 4\frac{n}{32} + \ldots + \log_2 n \frac{n}{2^{log_2 n}}$, proporcional a n.
- Como o trabalho é constante para cada troca, então o trabalho total da construção bottom-up é proporcional a n, melhor que a top-down.